$10 Volume XVI, Number 1 May 1994 June

FORTH DIMENSIONS

Interactive Embedded Development

Quicksort & Swords Redux

F83 Vocabulary Usage

Pygtools—Reusable Utilities

Generation & Application of Random Numbers

Contents

Features

6 ***Interactive Embedded Software Development*** — *Garth Wilson*

You can achieve complete interactiveness for developing Forth software on a target computer. The method requires no hardware or software except the host computer, the target, the text editor, and the metacompiler—no communications software, no emulator. And the metacompiler will only be used to compile code for ROM after it is developed and working.

13 ***Quicksort and Swords Redux*** — *Wil Baden*

After more than 30 years, C.A.R. Hoare's quicksort is still the fastest general algorithm for sorting in place on a single processor. Following up on earlier work, this FORML-award-winning author presents his implementation in ANS Forth (except for NOT), and shows how to implement in your Forth three words new in ANS Forth. And he narrowly avoids tempting fate...

21 ***Understanding F83 Vocabulary Usage*** — *Byron Nilsen*

Vocabularies are a unique feature of the Forth language. They provide a means to isolate groups of definitions so as to avoid naming conflicts, to preserve order in large applications programs, to reduce compilation time, and to achieve other worthy ends. We can ignore them at first, but eventually we will want to access words defined in other vocabularies, and when this need arises some confusion may follow. The discussion is for novices but, as another author points out in this issue, implementors have a different lesson to learn from it.

24 ***Generation and Application of Random Numbers*** — *Dr. Everett F. Carter, Jr.*

The world's computers generate ten billion random numbers per second. Many subtle problems can occur, and various compromises have to be made in order to even pretend to generate random numbers with a computer. This article explores the generation of random numbers and some important applications that use such numbers.

37 ***Pygtools—A Library of Reusable Utilities*** — *L. Greg Lisle*

One often hears of the need for reusable libraries of Forth tools. The most common call is for the ability to access C libraries from Forth. An alternative is a library structure designed for use with Forth, where the programmer can do whatever he or she wishes. In addition to that, this package demonstrates the flexibility of a screen-based disk structure. Written in Pygmy Forth, the concepts should apply to any Forth that provides block access.

Departments

4 ***Editorial*** ..... Interfaces and artifacts.

5 ***Letters*** ..... Malevolent fungus; Case of the human parser.

19 ***Advertisers Index***

42 ***Fast Forthward*** ..... Rapid development demands quality interfaces.

Editorial

Forth Dimensions
Volume XVI, Number 1
May 1994 June

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

Interfaces and Artifacts

I finally caved in to pressure from above and signed up for another on-line service. I had agreed in principle but resisted in practice—knowing these beasts can suck cash right out of the wallet (or the credit out of the line) and hours out of the day and night—until the free sign-up package arrived in the mail: mostly a diskette of custom software.

Custom software? What happened to twiddling my com settings until screen chaos and abrupt disconnects subsided to something passable? The new software would only work with the one service; good thing it included a generous number of free start-up hours, or my doubt might have triumphed and I'd still be using my generalized, tweak-when-you-change-services telcom setup.

Okay, it wasn't all dial-and-smile. My newfangled modem gave the software fits and this was, of course, not during business hours. But the tech-support voice mail worked surprisingly well, and ten or so minutes later I had acquired a different initialization string that had my modem shrieking cooperatively.

After that, I pretty much melted. I was cynical about the whistles (synthesized voice greeting at log-on), but loved the bells—an interface consistent with my own system's interface. Or, with another start-up diskette, consistent with the *other* popular interface, too, keeping lots of new users happier than they had expected to be. An interface might call attention to itself at first if we aren't used to it, but a good one makes it easier to focus on content and function. I've now seen both the net-jaded and the net-leery turn on to functionality dispensed via a decent interface.

Infer what you will about signs of the times, you can read more about the implications of interfaces in the new call for papers for this year's FORML conference: see page 12.

* * *

While rummaging around my new on-line service, I dipped into the newsgroups to see what was happening in the world of comp.lang.forth. What I found at the tip of the iceberg was pretty interesting: a list of the computer languages for which there are newsgroups, showing the current number of messages within each group. Especially in light of its grass-roots origins, Forth shows a decent ranking; besides, C and C++ should probably be discounted because of the institutional support they receive. (Pascal was once leader of the corporate pack, and where is it now?)

—Marlin Ouverson
ouversonm@aol.com

Admittedly Unscientific Sampling

5743	C++
5201	C
968	Fortran
932	Ada
668	LISP MCL
434	**Forth**
424	LISP
387	Modula-2
331	Eiffel
247	Miscellaneous
229	APL
221	Oberon
213	Dylan
199	IDL PV-Wave
170	Functional (functional languages)
115	Modula-3
91	ML (including Standard ML, CAML, Lazy)
86	Logo
34	CLOS (Common Lisp Object System)
2	Hermes (for distributed applications)

Forth Dimensions welcomes editorial material, letters to the editor, and comments from its readers. No responsibility is assumed for accuracy of submissions.

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $40 per year ($52 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, P.O. Box 2154, Oakland, California 94621. Administrative offices: 510-89-FORTH. Fax: 510-535-1295. Advertising sales: 805-946-2272.

The Forth Interest Group

The Forth Interest Group is the association of programmers, managers, and engineers who create practical, Forth-based solutions to real-world needs. Many research hardware and software designs that will advance the general state of the art. FIG provides a climate of intellectual exchange and benefits intended to assist each of its members. Publications, conferences, seminars, telecommunications, and area chapter meetings are among its activities.

"*Forth Dimensions* (ISSN 0884-0822) is published bimonthly for $40/46/52 per year by the Forth Interest Group, 4800 Allendale Ave., Oakland, CA 94619. Second-class postage paid at Oakland, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 2154, Oakland, CA 94621-0054."

Letters

Letters to the Editor—and to your fellow readers—are always welcome. Respond to articles, describe your latest projects, ask for input, advise the Forth community, or simply share a recent insight. Code is also welcome, but is optional. Letters may be edited for clarity and length. We want to hear from you!

Malevolent Fungus

Dear Mr. Ouverson,

God knows that I don't intend to write unreadable code. Indeed, after the code is written, it is eminently readable, logical, and sensible. But after aging for nine months or more in a cool, dark place, I am sometimes horrified to discover that the code has been transformed into an obscure Klingon dialect. Rays from an alien spacecraft? A malevolent fungus spawned in a toxic pond? Another Unsolved Mystery?

The only solution seems to be handing the code to a trusted Forth-literate colleague with instructions to question any aspect of the code. Your answers and explanations would then become the comments and documentation.

Garth Wilson's recent article, "Readability Revisited" (*FD* XV/6) presents many good ideas, but also inadvertently points out some differences between programming Forth screens and Forth text.

The need to program screens came with my Forth. For quite some time, I wrote in C programmers' mode, essentially taping screens together into a text scroll. I thought shadow screens wasteful and awkward.

After some experience with Forth, and a fifth read of Leo Brodie's *Thinking Forth*, I finally understood the how and why of short words with short stacks. It's interesting that Mr. Wilson, in commenting on Listing Two-a, shows a reluctance to factor the over-long code in `SAMPLEWORD` because the result would be a more dense and, hence, less readable text code.

After studying Charles Moore's code in cmForth, I've concluded that screens, unlike text, benefit from a high-density format. Because screens cause your eyes to focus at a small space, short words stacked with little white space are actually more readable and understandable than the looser style appropriate to text. Since white space in screens is comprised of real spaces, a dense format also compiles faster. Using shadow screens for comments makes sense now, as they don't clog the compiler but still can be printed adjacent the code with the six-screen-per-page format. What a circuitous journey!

FIG should be commended for reprinting *Thinking Forth*. While not a beginner's book, it is a must read for understanding Forth and Moore's philosophy.

Walter J. Rottenkolber
P.O. Box 1705
Mariposa, California 95338

Case of the Human Parser

Dear Marlin,

I applaud Garth Wilson's attempt in *Forth Dimensions* (XV/6) to improve the readability of Forth, but he hides some important advice near the end of his article and fails to follow it in most of his examples.

When I am reading a Forth program, I need one vital piece of information before I can understand it. Is what I am reading part of the code or a comment? In most of Mr. Wilson's examples, unless I visually parse a line to find the parentheses, I can't tell code from comment. One of Forth's strong points is that parsing is not needed to compile it. Why should it be necessary when one reads it?

Please, programmers, use upper case for code and normal, mixed upper/lower case for comments. Then one can tell at a glance whether one is reading something intended for a compiler to interpret or something intended for a human being to understand. One's mind has to change states between the two. Make the case match the reader's `STATE` flag.

Forth style is in the eye of the beholder. For example, I find Mr. Wilson's Listing Two-a merely confusing. There is no indication that it represents nested expressions unless one parses the code in detail. Listing Two-b, though under-factored, expresses the form of the code at a glance. Maybe this is one of those left-brain/right-brain issues.

Another grumble is prompted not by Mr. Wilson's style but by his Forth implementation. Just because BASIC used the fairly English-like syntax "`IF` test-condition `THEN` do-this-to-end-of-line," that is no excuse for any Forth to use the word `THEN` in the quite different syntax, "test-condition `IF` do-this `THEN`." In Forth, `IF` is an opening parenthesis. It marks the start of the expression to be executed if the condition test is true. It needs a related closing parenthesis. I am happy with `ENDIF` and I install it as an alias for `THEN` in any Forth I use. It might be more logical to use something like `IF{ do-this }ELSE{ do-that }IF`.

By the way, my programming environment sounds at least as simple as Mr. Wilson's. I use Mike Haas' Textra (written in J-Forth). Switching from editing to compiling is a matter of clicking the mouse in the J-Forth window or in the Textra window. I can have as many source-file windows open as I please. This is very handy for copying chunks from one's old programs. (PC users, don't try this at home. You'll need to buy an Amiga first.)

Regards,

Tom Napier
One Lower State Road
North Wales, Pennsylvania 19454

Interactive Embedded Software Development

Garth Wilson
Whittier, California

The purpose of this article is to describe how you can achieve complete interactiveness for developing Forth software *on* a target computer. It is very simple, and a natural for Forth.

The method discussed here is carried out with no hardware or software except the host computer (a PC-AT clone, in my case), the target, the text editor, and the metacompiler—no communications software, no µP emulator. You will probably want either a PROM programmer or a ROM emulator. The content of the ROM will be changed so few times during the development process that the time saved by a ROM emulator will be very little.

The metacompiler will only be used to compile code for ROM after it is developed and working.

The traditional idea of developing the software for an embedded system on a host computer first and then moving it over to the target has major weaknesses. The host often cannot handle or even emulate all the I/O your finished system will have; and even if it could, the timings and I/O port types would probably be different, forcing a painful transition. We will avoid that here.

Requirements

It is very easy to do the development on the target itself using target RAM, even though the final application usually goes into ROM. The target unit you use for development will need some things that you may be able to leave out of the end product, including:

A. the ability to interpret and compile Forth source code
B. a free port for inputting source code (bi-directional not necessary). I use an RS-232 port.
C. It should have some type of display, even if only an eight-character LCD. This will be the main feedback to the programmer as to the internal status of the computer.
D. A printer port is optional. I've always had one because it was so easy, but it is seldom needed. Sometimes it's nice for debugging.
E. appropriate port-driver software (for B and C above)
F. You may need more RAM for development than for the finished system.

The target computer can be virtually any computer with any processor for which a Forth kernel is available. The kernel can be either in ROM or loaded into RAM from a storage medium before beginning interactive development. This article primarily addresses the simple situation where the final code resides in ROM, and the same address space is used for program, data, headers, etc. Standard kernels may need some small changes in order to transparently transfer the already-developed code into ROM for the finished product. This will be addressed below.

Definitions

Before continuing, perhaps we should give some non-colon definitions. The "target" is a computer that we want to develop software for, but usually one that is inadequate for stand-alone software development use. This is often because it is an embedded computer with little or no user interface. This basically dictates that a host computer will take part in the development. The "host" is a separate computer involved in the development of the target's software. It is used for a varying range of functions, typically for writing, editing, and storing the source code and, later on, metacompiling the working finished code and programming it into PROM. The host does not have to be able to emulate the target's function.

The host is what most lay people think of when you say "computer"—it typically has a typewriter-style keyboard, a TV-style display, disc drives, etc. The target is typically an embedded computer, like the ones most lay people don't even know they own. Embedded computers lurk in TVs, VCRs, coffee makers, toys, under hoods, in car radios, printers, musical keyboards, microwave ovens, mobile telephones, and cameras, plus milling machines, plotters, extruders, film developers, navigation equipment, missiles... On and on the list goes.

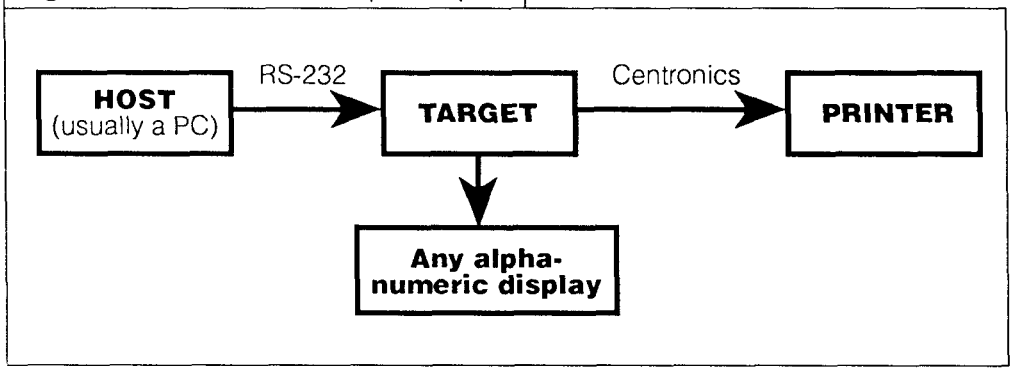
Getting Started

With an RS-232 or Centronics port on the target computer, a host computer such as an IBM PC clone can send information to it. Sending information the other direction is usually not necessary if you have even a small LCD or some other sort of display device on the target,

even if it is there for development only.

Since we don't need to send information back to the PC, we can let it think that the target is nothing but a printer. This paves the way to sending it a piece of code *without* leaving the text editor. It also means we don't need a terminal emulator or any other communications software.

Figure One. Hardware setup example.



We can let the target think the PC is nothing but an RS-232 keyboard, and we bring in the information using ?TERMINAL and KEY in EXPECT. In the systems I've done, I also had a small keypad that would be part of the finished system, so I wrote definitions that would do what we usually think of ?TERMINAL and KEY doing, calling them by the same names but preceded by a 'p' for 'pad,' that is, keypad—p?TERMINAL and pKEY.

A hardware setup example is shown in Figure One. All three links can be unidirectional.

In the Norton (programmers') Editor, you put your block markers (which on the screen look like little squares) around the code you want to send to the target, either by pressing "F4 S" or by clicking the middle mouse button when the cursor is where you want a marker. F4 is for block operations, and the S is for "Set block marker." The portion to send will appear highlighted on the screen.

Remember that since we're usually using EXPECT in the target, whatever we send it must be followed by a CR for the target to do anything with it. This means your second block marker will usually be on the left margin of your PC screen, on the line immediately after the last line you want to send. To actually send it from the Norton Editor, press "F7 B Y". The F7 is for printer operations; the B is for "print Block"; and the Y is for "Yes," since it asks, "Print block?"

I set it up to use a serial port from the PC. Normally, printer output from the text editor is directed to the parallel port, but it can be redirected in a DOS MODE instruction in a batch file. I have called the batch file "TARGET.BAT," and it consists of:

```
C:\DOS\MODE COM2:9600,N,8,1,P
C:\DOS\MODE LPT1:=COM2:
```

The first line sets up the PC port for (in this case) 9600 bps, no parity bit, eight data bits, and one stop bit. The "P" tells it to continually re-try if the target is not ready to accept data.

My batch file to undo the port redirection just says,

```
C:\DOS\MODE LPT1:
```

I must admit that, unfortunately, this did not always work on one of our computers. It acted happy to do it, but sometimes nothing happened. (Someone ought to write a song about DOS. More on that in the sidebar, "Beware of DOS.")

Since you're using a text editor, you might as well keep the number of data bits at eight, not seven, because with the text editor it is easy to use characters that are not on the keyboard, like °, Σ, μ, Ω, and others that will be very useful in typical embedded applications. You can also use the graphics characters to put diagrams in the comments in your source code. Most of these special characters have values between $80 and $FF, and they will need the eighth data bit. Note that I'm not talking about word processors that put in all those unprintable control codes that are hostile toward compilers.

These characters usually won't show up correctly in the intelligent LCD modules that everybody and their brother makes now, since they usually interpret the upper 128 values as Japanese characters; but you will get the benefit of nicer source code, and you wouldn't want FIND to confuse "½" and "+" in word names just because they look the same when you strip off the high bit. The only drawback is that Forth words that find an NFA given a CFA or LFA (like L>NAME) may not always work right without modification. I used the high bit in names for over a year before I finally modified my L>NAME—it's just not a big deal unless you depend heavily on 100% correct output of WORDS and decompiling words.

The serial port on the target will get set up by INIT-HDWR in ABORT. I have implemented a 256-byte software buffer and zero-overhead, high-level Forth interrupt response. Before I figured out how to accomplish the interrupts in an F83 system that didn't come with any such support, I just had ?TERMINAL poll the serial interface IC. That works, but it's slower, which can be an issue if you want to send more than a few lines at a time to a very slow target.

The zero-overhead interrupt support is very simple, adds only a few bytes to your overall code, and can be nested as many interrupt levels deep as you wish, as long as you don't run out of room on your data or return stacks. Even separate stacks are not necessary. I'll get to this in another article. It's another natural for Forth.

I made TIB to be 128 bytes so it can handle any line I send it, even if it hangs over the right edge of my 80-column PC screen a little ways.

The sidebar entitled "Beware of DOS" tells of some of the DOS problems I ran into in the effort to get the PC talking nicely to the target. It may save you a lot of time if you use a DOS machine.

Once we have our interactive system up, we can write a piece of code (optionally followed by a line to try it out),

```
: HELLO    ( name ( -- )
   CR ." Hello "
   BL WORD COUNT TYPE
   ." !  How are you?"    ;

HELLO Mac
```

put our block markers around it to send it out to the target (remembering to put the last block marker after the last line's CR so EXPECT knows to wrap it up so it can be passed on to INTERPRET)

```
■: HELLO    ( name ( -- )
   CR ." Hello "
   BL WORD COUNT TYPE
   ." !  How are you?"    ;

HELLO Mac
■
```

and send it (by pressing "F7 B Y" if you are using the Norton Editor). The target should ignore LF characters. As I'm writing this, I tried the above to be extra sure I didn't have some silly embarrassing mistake. After I pressed the "Y," I turned toward the workbench and my target had already compiled HELLO and executed the last line, leaving "Hello Mac! How are you? ok{ 0 }" in the display.

If you wanted to FORGET the definition and recompile it with changes, you could just type FORGET HELLO and put the first block marker above or to the left of it before sending it:

```
■FORGET HELLO
: HELLO    ( name ( -- )
   CR ." HI "
   BL WORD COUNT TYPE
   ." !  I like your tie."    ;

HELLO Mac
■
```

After sending it, we get "Hi Mac! I like your tie. ok{ 0 }" in the display.

So what happened? Actually the same thing that would have happened if the target had a full user interface and you could have typed the same lines into the target at several thousand words per minute without mistakes! However, since you first typed it in on your text editor, you have the source code there, immune to target crashes even if you had not saved it to disc before sending it out. The colon definition also got compiled into the target's RAM and has become part of the target's Forth.

Actually, I would recommend that you save your source code to disc before sending out a block if you don't know for sure you have everything hooked up and working, since if there's a communication fault, you'll be given a message (again, compliments of DOS) that says something like "Drive not ready" or "Out of paper" or "Write fault," followed by the choices "Abort to DOS, Retry, Ignore?" Notice that none of the choices is to abort just the print and go back to your application, in this case the editor! Abort loses any new stuff, retrying will get you nowhere unless you leave the message there until you are able to correct the problem; and ignore is only good for one byte, then you get the beep and the error message again.

Since the interactive development we're after depends on a certain minimum amount of code on the target already working, we will have to get that part working first by using our old-fashioned methods. Fortunately, it's not that much (except when you have the misfortune of running into problems like the ones described in the sidebar "Beware of DOS"). You can soon get on to developing your application with total interactiveness and almost instant turnaround time between writing a piece of code and trying it on the actual target.

The turnaround time here is the amount of time required for you to place the block markers where you want them in the text file (typically with the mouse), then press the appropriate key or keys to "print" it, and whatever time it takes the target to process the incoming information. As I set it up, everything taken in by EXPECT also gets displayed.

In the June '93 issue of *Electronic Component News*, Randy Devol of Nohau Corporation has an article on emulators. He says there that the "traditional" change-compile-test method with EPROMs has a cycle time of five minutes to an hour, and goes on to say that with their emulators you can bring the cycle time down to as little as ten seconds. Although the interactive development method I am advocating here does not by itself provide the instruction history that the emulator does, it makes it possible for you to get through the change-compile-test cycle even faster than the ten seconds Mr. Devol is touting. This is one of the things that got me hooked on Forth.

Even the slowest targets should be able to finish a typical line in a second or two. If this seems terribly slow, consider that you could be testing a typical definition less than ten seconds from when you're finished writing it, and without leaving, suspending, or backgrounding the editor. Most targets will be much faster. I have done this on 2 MHz 65C02's, which would handle the source code coming in at about 250 bytes per second for normal compiling. This includes displaying the lines sent over, and searching through a 500-word vocabulary for compilation. It would go considerably faster if the source code were to come in on a parallel port instead, since less time would be taken actually operating the port. The physical limit on a 9600 bps serial port would be about 960 bytes per second, since it takes ten bits to get a byte through—one start, eight data, and one stop.

On a slow target, it's nice to have a way to make changes in previously compiled code without recompiling. This is to save the time of recompiling all the code that follows it and depends on it. I have defined a few words I occasionally use to do this.

If you're thinking ahead, you might have already thought of the situation where you have a lot of code

compiled in RAM and you crash the system. Unless you have a lightning-fast target computer, it's going to take some time to recover from the crash because of all the code you will have to recompile. In my experience, most crashes are due to getting into loops where the exit conditions are never met, and they seldom destroy data. My five-second, crash-recovery method is to press the reset button, then have the cold boot routine ask, "NEW?" to which I respond by pressing the "no" key. This avoids reinitializing the dictionary pointer and CONTEXT. Immediately, the target computer is back under control as if it had executed ABORT instead of an uncontrolled crash. There is usually no loss of data or program.

There are also times when I find a problem with something that got put into the ROM a long time ago, and there's a lot of stuff after it. Re-defining it is, of course, no problem, so I might work on it in RAM again, then do another EPROM sometime after I believe the bug is taken care of. When you accumulate a lot of code working in RAM, you will want to metacompile to put it in ROM (an erasable one or a ROM emulator, since we're in the development process).

Figure Two shows a typical software diagram of the flow of code during and after code development. Only the top line (labeled "Forth source text") is used during actual development. The other paths are for getting the code into

Beware of DOS

Forth Dimensions recently published a letter I wrote to the editor, which I started by commenting on an article written by Russell Harris entitled, "A Lesson in Economics" (*FD* XIV/5).

Harris tells of problems with using PCs in embedded and instrument-control applications, especially real-time. I praised the article, but was embarrassed later to realize that I made myself sound like an over-zealous, biased neophyte who really didn't know, and that I thought everybody who embedded a PC was naïve. What I had failed to mention is that I've had many of those problems myself, and that it's nice to have several articles to be able to prove that it's not just me. I've wasted about six months of my working time in the last six years just to DOS and DOS-machine problems that had no excuse for existing. Unfortunately, I have to be compatible with the rest of the industry, so I have a DOS machine on my desk.

One of the nasty problems I've run into goes something like this: you send data out a PC's serial port. Sometime right before the target finishes receiving the last byte necessary to fill its buffer, the target sets the CTS line false, whereupon the PC should just finish the byte already started, then quit. The problem is that sometimes it will send out a few more bytes before realizing that it's supposed to stop.

This is something that drove me nuts for a while, even after figuring out what was happening and being able to see those bytes frozen on a DSO. The problem was worse on our MS-DOS than on DR-DOS. When I bought the DOS computer I use at home, it came with DOS 5.0 installed, but I had to go back to DOS 3.3 because my editor would not recognize my mouse under DOS 5.0. I don't know if the serial port problem would exist under later versions of DOS. I may have to buy a newer version of the editor.

Oh, the joys of DOS! In the recent words of a co-worker, "DOS ist *not* gooht!" I tried using different hardware handshake possibilities to see if the PC would respect DSR any more than it does CTS (or vice versa, or both together), but I was not able to completely get rid of the problem. I haven't looked into whether I could write my own port driver for the PC and expect the editor to use it when it tries to "print." It may require writing my own BIOS, which does not interest me.

My main remedy was to have the target keep the CTS line false starting when there were six bytes left in the buffer to fill. I first tried two and then three bytes, but it wasn't enough. Before I went to hitting the brakes with six bytes left to go, out of desperation I had also tried slowing the baud rate way down. Although this made it easier for the target to keep up with the data coming in, strangely enough it also increased the chance that the PC wouldn't quit right away those times that it did receive a CTS-false signal.

Obviously, having the interrupt support on the target is helpful, since otherwise the data sent when the target is saying it's not ready would be lost. If you're a DOS expert (and I'm not convinced there is any such thing), I would appreciate an explanation for this and other problems.

While we're on this note, I suppose I should warn you of another related PC problem I encountered, in case the warning might save you some time. Since I did not have interrupts yet the first time I implemented this idea of interactive development on the target, I would pulse the CTS line of the RS-232 interface long enough to get a byte started, but not long enough to get more than one byte through. I made the pulse about half as long as the amount of time required to get a whole byte through. After giving enough time for a byte to finish coming in, I would poll the ACIA to see if, indeed, a byte did come in. But I wrongly assumed that if a byte did come, the PC would have started it some time at, or before, the time CTS went false.

What actually happened was that, if the next byte for the PC to send was the first one of a new line, the PC might wait up to ten or fifteen milliseconds before starting that byte. (Keep in mind that a byte at 9600 bps only takes about one mS to transmit!) The solution for that was simply to wait longer before polling the ACIA if SPAN was still zero. Of course, this slowed things down; but without doing this, the first byte might finish up during a later pulse, and that pulse would start the second byte, some time after which you poll the ACIA which tells you there is, in fact, something to be picked up. The second byte is read, appearing to be the first in the line. The real first byte got run over by the second, and was lost. This may have been an idiosyncrasy of my text editor. I don't know. It was all too strange—my com port problems were sometimes different if using, for example, the DOS PRINT command.

ROM space after development.

Transparent Transfer to ROM

Now, we wouldn't totally be accomplishing our goal if, after getting our code working, we had to modify it for the metacompiler to accept it so we could get it into ROM. You may have to change some definitions or some methods to make sure the code you got working in RAM will make the transfer to ROM transparently.

A problem I had to address with a metacompiler I was using a couple of years ago was the fact that the metacompiler would not allow (for example):

```
[ YR MINUTES - 1+ ] LITERAL
```

where YR and MINUTES were variables. It would stop and tell me it couldn't access a ROMed variable. The explanation in the manual was that the metacompiler is trying to defend against operation in the target's RAM area, since it does not exist (or is not accessible) at metacompilation time. This makes sense; but the job of a variable is to put the first address of the variable or array on the stack, and in this case all I want to do is compile another number (a byte count) that is pre-calculated from other variables' addresses.

The vendor's solution was to use >DATA as follows:

```
[ ' YR >DATA  ' MINUTES >DATA - 1+ ]
                                LITERAL
```

Since this was so cumbersome, I redefined VARIABLE to be a constant that just returned the address of the next available RAM byte. This still didn't work, because the metacompiler ignored redefinitions of essential words like VARIABLE. My final solution (which ended up working very well) was to rename it VAR. I also made it require an input, the number of bytes I wanted in that variable. I know this breaks away from the Forth-83 Standard, but it helps the transition from RAM to ROM because the first byte of the variable's data area is the same whether the constant pointing to it is in the ROM or right after the data area in RAM. The definition is

```
: VAR       ( name ( n -- )
   THERE SWAP ALLOT CONSTANT    ;
   IMMEDIATE
```

The IMMEDIATE was a requirement of our metacompiler to interpret it. A typical syntax example of our new variable word would be

```
6 VAR TMBUF   ( TIME BUFFER W/ ROOM FOR  )
              ( SEC, MIN, HR, DAY, MO, YR)
```

In a ROMed system, if you have five variables in a row that each have a two-byte data field, those data fields will be one after the other in RAM. This makes it nice if you want to initialize the whole group of them with a single CMOVE or ERASE, for example. However, when you develop in RAM, the variable's headers and code will sit between those data fields. You can get around this difference if you want to by writing something like what is shown in Listing One, which will work with both systems. (This particular example is from the automated test equipment system I did a couple of years ago. At the end of a set of tests, the information in RESULTS got moved to the end of the archive data chain, if archiving was requested.)

Other changes I had to make were due to bugs in the software we bought. Since the target source code was supplied (basically right out of the public-domain stuff), I was able to fix the target-specific bugs. I've just had to learn ways of getting around most of the metacompiler bugs, since the metacompiler is a black box to us, the customer. I'd be glad to try to help anyone who suspects they may be having the same problems, but I don't want to use this space to publicly dishonor the supplier. We got good use of the software, and I intend to buy another metacompiler from them.

As I pointed out earlier, this is all done with no hardware or software except the host computer, the target, the text editor, and the metacompiler—no communications software, no hardware emulator. In my experience with embedded systems, the emulators you really need would be for the I/O silicon anyway. I've spent a lot of time trying to guess what's going on inside peripheral interface ICs whose data books didn't tell me all I needed to know.

There are times, however, when you would like to put in a breakpoint to stop and look around at the stacks, different variables, etc., for debugging purposes. Listing Two shows a word BREAK for that purpose. I seldom need it, but it is valuable when the need arises. It can be called anywhere in your code where you might want a breakpoint. The code should be pretty self-explanatory. Notice that you can even compile new code or make changes, since this is basically the same as what's in QUIT. The fact that

Listing One

```
      100 VAR RESULTS    RESULTS  ( Test results array.  1st 6 bytes are time & )
  DUP CONSTANT ARCTM        5 +   ( date simply moved from variables MINUTES-YR.)
  DUP CONSTANT OP#          1 +   ( Next is a 1-byte operator number,           )
  DUP CONSTANT MODEL#       2 +   ( 2-byte model identifier number [actually the)
                                  ( address of string of test menu item name.]  )
  DUP CONSTANT MAXTEST#     1 +   ( Highest test number logged.                 )
  DUP CONSTANT PASSf        2 +   ( 2-byte pass flag [true=pass, false=fail.]   )
      CONSTANT 1STRESULT          ( Start logging results here. Test number will)
                                  ( be used as index for !ing & @ing results.   )
```

it suspends other definitions that were in the process of executing is okay, as long as you don't mess up the stack or any critical values before exiting BREAK.

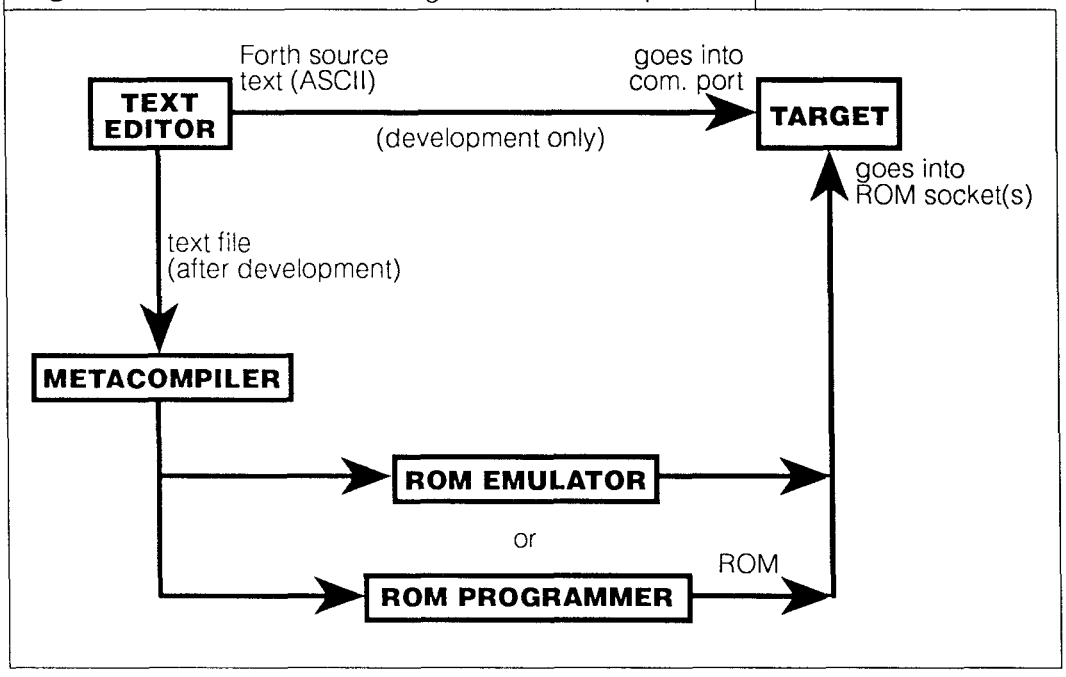
Assembly Development

Normally, the target will not have any assembling capability. However, this interactive development method can still be used to greatly speed up the development of primitives by using Forth definitions for what amounts to a monitor program. This "monitor program" can have seamless integration with the Forth system, because it *is* part of the Forth system. This way, you can use your favorite macro-assembler and linker to generate the machine-language code from the assembly-language source code, then feed Intel hex (or whatever you like) over the RS-232 into the target. Making a change in your assembly-language source code and trying it out in your Forth application still gives a relatively fast turnaround time.

I use a word LDMEM (load memory) to bring the Intel hex file from the assembler into the target. After LDMEM is invoked, it receives the hex file, putting the data bytes into memory and checking for errors according to the Intel hex protocol. After the last line (:00000001FF), LDMEM is finished and the target will again expect Forth text, just as before.

A separate assembler and linker that is not related to your Forth software will be very clumsy at writing primitives that would be made part of the target's Forth. To remedy this, you can have a primitive that just calls the subroutine at the address pointed to by a variable. I used a small assembly language routine for digital audio playback in a recent project. After I was happy with it, I didn't even bother to rewrite the assembly language code so the metacompiler could use it. I just relocated it to ROM space and merged the hex output files from the assembler and the metacompiler. It made the combination painless, and it allowed me to use a nice macro-assembler.

Figure Two. Flow of code during and after development.



Conclusion

Recently, I was talking to a friend who used to work with me, whom I respect very much. He had gone to work at another company, where his experience in C and the 68HC11 was a perfect fit. As we were talking about our development

(Continues on page 23.)

Listing Two

```
: BREAK          ( -- )                    ( This is a code-insertable breakpoint)
   KEYBEEP                                 ( Announce arrival at breakpoint.      )
   BEGIN                                   ( Loop until user wants to exit BREAK.)
       CR   ." >"          CURSORON        ( Display the prompt & flash cursor.   )
       QUERY    >IN OFF    CURSOROFF       ( Get ready to process input line.     )
       SPAN @ 1 >                          ( Empty line [CR only] means to exit.  )
       IF   INTERPRET                      ( Line not empty: hand to INTERPRET.   )
            STATE @ 0=                                ( If not in compile mode,      )
            IF   ." Bok{"    DEPTH 0 .R    ." }"      ( give the "ok" & the stack    )
                 WAIT1/2SEC                           ( depth with enuf time to see,)
            THEN FALSE                       ( & signal to UNTIL not to exit BREAK.)
       ELSE CR ." LEAVING BREAK"             ( If input line empty, announce exit,  )
            ATTNBEEP WAIT1/2SEC              ( beep, give time to see announcement,)
            TRUE                             ( then signal to UNTIL to exit BREAK.  )
       THEN
   UNTIL                  ;
```

1994 FORML Forth Conference

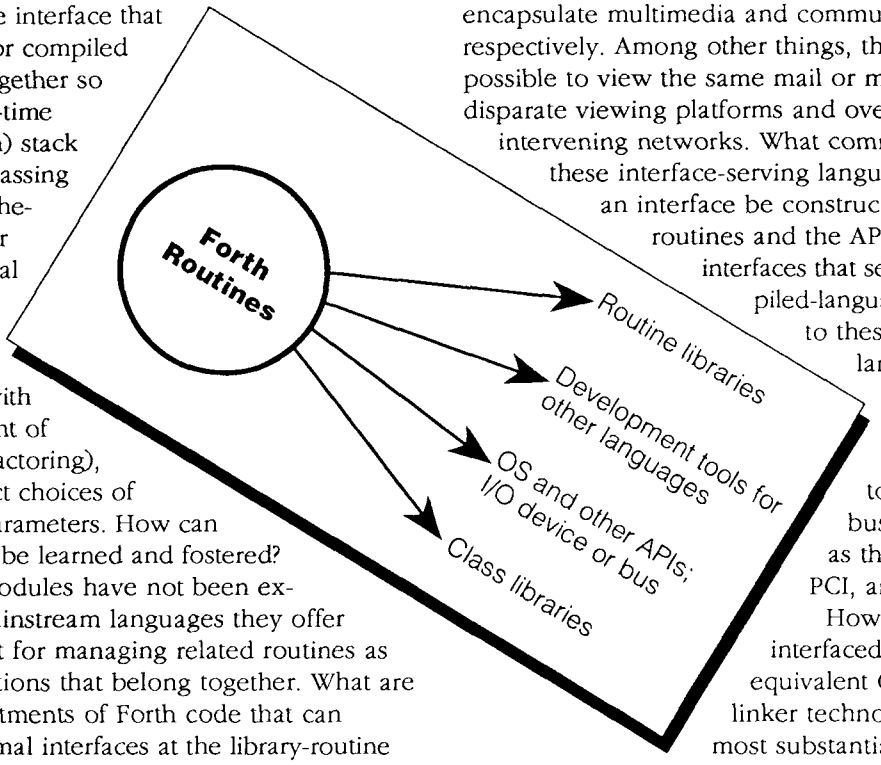
Conference Theme:

"Interface Building"

Call for Papers

Papers are sought that explore how code and data resources in various forms can be interfaced to maximize code reuse and programming efficiency.

Compiled routines represent the most fundamental code resources. The interface that makes it possible for compiled routines to work together so well involves a run-time system's call (return) stack and its parameter-passing mechanism. Nevertheless, exploiting their cooperative potential requires skillful programming. Each routine must be outfitted with just the right amount of functional scope (factoring), and with the correct choices of input and return parameters. How can this *interfacing art* be learned and fostered?



Libraries and modules have not been exploited well. In mainstream languages they offer only token support for managing related routines as (indivisible) collections that belong together. What are some possible treatments of Forth code that can establish more formal interfaces at the library-routine level or the module level?

Can interfaces be fashioned between Forth routines and the libraries, run-time systems, or data structures of other languages?

New programming languages keep appearing to tame various interfacing problems. Examples include Postscript, which establishes an interface around diverse printing engines so they can be treated similarly. Open Firmware (formerly Open Boot) wraps a standard environment around computer sub-system components, facilitating their configuration and initialization. X-Script and Telescript encapsulate multimedia and communications services, respectively. Among other things, they make it possible to view the same mail or multimedia item on disparate viewing platforms and over disparate, intervening networks. What common features do these interface-serving languages possess? Can an interface be constructed between Forth routines and the APIs and system call interfaces that serve as the compiled-language counterparts to these interface-serving languages?

Can Forth modules be crafted to let it talk to one or more I/O bus interfaces, such as those for PCMCIA, PCI, and "Plug N Play"? How can Forth be interfaced to Windows, or equivalent GUIs? Besides linker technology, what is the most substantial obstacle that prevents our use of GUI-encapsulating class libraries such as MFC or OWL? Because SOM (system object model) attempts language independence, can it lead to a Forth interface to class libraries? What run-time interface provisions besides a call stack and a parameter-passing mechanism are going to be needed to support object-oriented Forths? To support event-driven programming?

Time allotments for presentations will favor early submittals and/or theme relevance. Abstracts are needed by September 1, 1994.

Mail your submissions to:

Forth Interest Group
att'n: Mike Elola
P.O. Box 2154
Oakland, California 94621

Quicksort and Swords Redux

Wil Baden
Costa Mesa, California

"Quicksort and Swords" was the first Forth-83 program to be published in *Forth Dimensions*. A decade later, here they are reconsidered for Standard Forth, *a.k.a.* ANS Forth.

Quicksort

In this implementation of `quicksort`, except for `NOT`, all constituent words are Standard Forth core or core extension words. In the unlikely event that you don't have `NOT` as a primitive, replace it with the ugly `0=`. Three words new in Standard Forth, `ALIGNED`, `CELL+`, and `CELLS`, can be implemented easily in your Forth.

This implementation is not recursive and thus avoids the overhead of recursion. Improved code makes it even a little faster. Freedom from the Forth-83 constraints of blocks of 16 rows and 64 columns, and everything in upper case, makes it easier to program and comprehend. With Forth's stack no data structure has to be defined.

Although the code is longer in column inches, it has fewer characters than two old fashioned blocks, and is about the same compiled size.

After more than 30 years, C. A. R. Hoare's `quicksort` algorithm, with slight modifications from sundry authors, is still the fastest known general algorithm for sorting in place on a single processor. Naming something "quick" or "fast" invites the fates to humble you, but `quicksort` is still quick.

The algorithm works like this.

If an array has more than a *certain number* of elements, *partition* the array into two sections such that no element of one section comes after any element of the other section.

But if the array has at most the *certain number* of elements, then use a *simpler method* with less overhead to sort it.

Keep doing this with the resulting sections until there are no sections left.

This begs the questions:

What is the certain number?
What is the simpler method?
How do you partition?

For the original Hoare algorithm the *certain number* is 1, and the *simpler method* is "don't do anything."

For many implementations in profane languages the *certain number* is something between 7 and 17, and the *simpler method* is "insertion sort."

For this implementation the *certain number* is 3, and the *simpler method* is "just do it."

To *partition*, the ideal thing to do would be to take the median element and put everything on one side of it in one section and everything on the other side in the other section.

(The median is the element that will be in the middle when the array is sorted.)

But you don't know what or where the median element is.

So you guess.

And how do you guess?

Hoare said in effect: Since you can't tell, take an element at random—nondeterminism will give optimum performance; if that's too much trouble just take the first element, although with extremely bad luck such as everything being in order or almost in order already, this will be as bad as bubble sort.

A less mystical method that has been found to be practical is: Take the first, last, and middle elements; arrange them in order with two or three comparisons; and take the median of the three as your guess for the median of the whole.

That's what is done here.

This has a further pay-off when we do the actual partition, as we shall see.

As this is Forth, the code is written bottom to top, but the comments here are top to bottom.

I'm not going to tempt fate by calling the top word `QUICKSORT`, especially since I know some ways to make it faster, but I call it `QSORT`, hoping that the fates don't recognize acronyms.

`QSORT ( a-addr k xt -- )`

Take an aligned address `a-addr`, a count of the number of elements `k`, and an execution token for a comparison routine `xt`: sort the `k` cells at `a-addr` with the comparison routine given by `xt`, using the method of quicksort.

(Code follows; text continues on page 16.)

```
 1 ( Hoare's Quicksort )( Non-Recursive )( Wil Baden 1967-1993 )

 3 variable    INORDER#

 5 : EXCHANGE  2dup  @ >r  @ swap !  r> swap ! ;        ( x y -- )

 7 : OrderThree                          ( lo hi mid -- lo hi mid )
 8       >R                                          ( lo hi)
 9             over @  R@ @  INORDER# @ execute  0>
10                   if    over R@ EXCHANGE    then
11             R@ @  over @  INORDER# @ execute  0> if
12                   R@ over EXCHANGE
13                   over @  R@ @  INORDER# @ execute  0>
14                         if    over R@ EXCHANGE    then
15             then
16       R>                                          ( lo hi mid)
17 ;

19 variable    guess

21 : SkipLowers                          ( x y -- x y )
22       >R                                          ( x)
23             begin
24                   cell+
25                   dup @  guess @  INORDER# @ execute  0< not
26             until
27       R>                                          ( x y)
28 ;

30 : SkipHighers                         ( . y -- . y )
31       begin
32             1 cells -
33             guess @  over @  INORDER# @ execute  0< not
34       until
35 ;

37 : Partition                           ( lo hi -- lo y x hi )
38       2dup  over -  2/  aligned  +                ( lo hi mid)
39       OrderThree
40       @ guess !                                   ( lo hi)
41       2dup                                        ( lo hi x y)
42       begin
43             SkipLowers
44             SkipHighers
45             2dup > not
46       while
47             2dup EXCHANGE
48             2dup  2 cells -  >
49       until
50             >r cell+ r>
51             1 cells -
52       then
53       SWAP ROT                                    ( lo y x hi)
54 ;
```

```
: SmallerSectionFirst                ( lo y x hi -- lo y x hi)
     2over 2over  swap -  >r  swap -  r>  <
           if    2swap    then
;

: Hoarify                            ( lo hi -- ... lo hi)
     begin ( `SortSnapshot' goes here. )     ( ... lo hi)
           2dup  swap -  2 cells  >
     while
           Partition                          ( ... lo y x hi)
           SmallerSectionFirst
     repeat                                   ( ... lo hi)
;

: OrderAPair                         ( lo hi -- )
     2dup = not if
           over @  over @  INORDER# @ execute  0>
                 if    2dup EXCHANGE    then
     then                                     2drop
;

: ShortOrder                         ( lo hi -- )
     2dup  swap -  1 cells  > if
           dup  1 cells -                     ( lo hi mid)
           OrderThree
           drop 2drop
     else
           OrderAPair                         ( )
     then
;

: QSORT                              ( a-addr k xt -- )
     INORDER# !                               ( a-addr k)
     dup 2 < if    2drop exit then
     1-    CELLS    OVER +                    ( lo hi)
     DEPTH >R
           BEGIN                              ( ... lo hi)
                 Hoarify
                 ShortOrder                   ( ...)
                 DEPTH R@ <
           UNTIL                              ( )
     R> DROP
;

( Function to compare counted strings. )

: CCOMPARE                    ( c-addr-1 c-addr-2 -- flag )
     >R  COUNT  R>  COUNT COMPARE
```

If there are no elements, there's nothing to sort.

It's more convenient for the program to work with first and last elements than address and count, so I convert.

DEPTH is saved on the rack to tell when there are no more sections.

QSORT uses Hoarify to make partitions until it makes a partition with no more than three elements. This small partition is sorted by ShortOrder.

When there are no more partitions, we're done.

ShortOrder checks whether the section has three, or fewer, elements. Three elements are sorted by OrderThree using brute force.

With two elements OrderAPair makes a comparison to see which comes first. With one element OrderAPair doesn't have to do anything.

So long as it is looking at a section with more than three elements Hoarify uses Partition to turn that section into two sections.

SmallerSectionFirst is used to select the smaller of the two sections for the next use of Partition. This is insurance against stack overflow.

Partition is the essence of Hoare's method. As mentioned above, we take the first, last, and middle elements, and put them into order using OrderThree. The median of the three is used as the guess of the median of the whole section.

Elements at the beginning of the section which are lower than the guess are skipped using SkipLowers. Elements at the end of the section which are higher than the guess are skipped using SkipHighers. Until the skippings cross over each other we EXCHANGE the two elements that have terminated SkipLowers and SkipHighers.

Because we have taken the median of three as the guess, SkipLowers and SkipHighers are each guaranteed to find an element to cause termination of the search. Thus just one test is needed. Also we can begin testing with the second element.

OrderThree arranges three elements in order by making two or three comparisons and exchanges.

EXCHANGE takes two addresses and exchanges their contents.

INORDER# is the execution vector containing the execution token of the Forth word that decides the order of two elements.

Swords

WORDS is a Standard Forth word that cannot be defined with Standard Forth words. None-the-less it generally comes with source using some of your implementation's non-standard extension words. Since I don't know your implementation I will describe in detailed vagueness what you must have for WORDS to work.

- You must have a way to get a word identifier for the most recently defined word in your current environment. With elaborate dictionary structures this may not be an easy thing to do. The most useful word identifier would be an execution token.
- You must have a way to go from a word identifier to a pointer to a string naming the word. In Forth-83 this was >NAME. This also may be non-trivial in some implementations.
- You must have a way to print the name of the word, preferably with some simple formatting.
- You must have a way to go from a word identifier to the word that was defined just before it. Again in some implementations this may be tricky. We impose the requirement that the last word identifier to be printed return 0 for this operation.

From the actual definition of WORDS in your system you can scavenge what you need to define SWORDS, "sorted words."

I'll take the definition of WORDS in This Forth as a example of one of the simplest definitions.

```
: WORDS
      depth 0= ?? LAST   ( xt)
      0 COL# !
      begin
            ?DUP
      while
            dup .ID
            >PREVIOUS
      repeat              ( )
;
```

There's an alien word "??" but the intent of the line is clearly to give us a word identifier to start our listing.

0 COL# ! must be to initiate formatting.

```
: .ID      >NAME .WORD ;
```

The definition of .ID gives us >NAME and .WORD. With >PREVIOUS these look like the operations we want.

Their implementation in This Forth use some more unexplained extensions and definitions. They will be different in your system.

```
: >LINK   2 - ;
: LINK>   2 + ;
: LAST    0 has has LINK> ;
: >NAME   1- has ;
: .WORD          ( c-addr -- )
    count              ( c-addr len)
    dup MORE
    type               ( )
    TAB
;
: >PREVIOUS    ( addr -- addr )
    >LINK has dup ?? LINK>
;
```

Let's whack the definition of WORDS, replacing the occurrence of .WORD with something to save a value in an array rather than print what the value is pointing to. We can then sort the array with QSORT, and finally print the array.

The implementation factors we'll use are:

Quicksort in Action

As an example of QUICKSORT in action, here are snapshots of a short sort.
Arrange the 20 most frequent words in English in alphabetic order. We start with them in order of their frequency.

```
[ the of and to a in that is was he for it with as his on be at by i ]
```

Take the first, last, and middle words, and guess "i" as the median. Using this gives us two equal-sized sections.

```
[ for by and at a be his as i he ] [ was it with is that on in to of the ]
```

With equal-sized sections take the upper one next. Guess "the" as the median.

```
[ for by and at a be his as i he ] [ on it of is that in ] [ the to with was ]
```

In the shorter section guess "was".

```
[ for by and at a be his as i he ] [ on it of is that in ] [ the to ] was [ with ]
```

No guessing with at most three elements.

```
[ for by and at a be his as i he ] [ on it of is that in ] [ the to ] was with
[ for by and at a be his as i he ] [ on it of is that in ] the to was with
```

Guess "is".

```
[ for by and at a be his as i he ] [ in is ] [ of it that on ] the to was with
[ for by and at a be his as i he ] in is [ of it that on ] the to was with
```

Guess "on".

```
[ for by and at a be his as i he ] in is [ of it ] on [ that ] the to was with
[ for by and at a be his as i he ] in is [ of it ] on that the to was with
[ for by and at a be his as i he ] in is it of on that the to was with
```

Guess "for".

```
[ be by and at a as ] [ his for i he ] in is it of on that the to was with
```

Guess "his".

```
[ be by and at a as ] [ he for ] his [ i ] in is it of on that the to was with
[ be by and at a as ] [ he for ] his i in is it of on that the to was with
[ be by and at a as ] for he his i in is it of on that the to was with
```

Guess "at".

```
[ as a and ] at [ by be ] for he his i in is it of on that the to was with
[ as a and ] at be by for he his i in is it of on that the to was with
```

All done.

```
a and as at be by for he his i in is it of on that the to was with
```

Appendix

Stock Definitions—Common Implementation Factors

```
variable COL#
variable COLS#    72 COLS# !
variable TAB#     8 TAB# !

: LINE   CR    0 COL# ! ;
: MORE   dup  COLS# @  COL# @  -  > ?? LINE COL# +! ;
: cols   COLS# @  COL# @  -  min   0 max   dup SPACES   COL# +! ;
: TAB    COL# @  TAB# @ mod  TAB# @ swap -  cols ;
```

Sample Output from SWORDS

```
!          "          #          #>         #s         $          '          (          (.)
*          */         */mod     +          +!         +loop      ,          -          -->
.          ."         ..        .id        .r         .s         .word      .words     /
/mod       0x         1k        2!         2/         2>r        2@         2drop      2dup
2over      2r>        2r@       2rot       2swap      3dup       4dup       :          :noname
;          ;s         <         <#         <>         =          >          >body      >char
>link      >name      >previous            >r         ?          ??         ?do        ?dup
@          Hoarify    OrderAPair           OrderThree            Partition
ShortOrder            SkipHighers          SkipLowers            SmallerSectionFirst
[          [']        [0x]      [char]     [here]     \          ]          abort      abort"
abs        again      align     aligned    allot      and        andif      argument
base       begin      bl        bye        c!         c"         c+!        c,         c@
case       ccompare              cell+      cells      char       char+      chars      close
closed     col#       cols      cols#      compare    constant              count      cr
create     d+         d-        decimal    depth      do         does>      drop       dup
elective              else      emit       empty      eol        esac       evaluate
exchange              execute   exit       false      file       fill       filter     find
flush      flushed    guess     has        here       hex        hold       i          i'
if         immediate             inline     inorder#              input      invert     j
k          key        last      leave      line       link>      literal    load       loop
lshift     marker     max       min        mod        more       move       negate     nesting
nip        not        of        open       opened     or         orif       ount       output
outside    over       pad       page       parse      patch      pick       please     postpone
push       qsort      quit      r>         r@         recurse    refill     reinput    reopen
reopened              reoutput             repeat     rewind     roll       rot        rshift
s"         s+         search-wordlist      seek       sign       sought     source-id
space      spaces     state     stream     swap       swords     system     tab        tab#
tell       th         then      time&date            to         told       true       tuck
type       u.         u.r       u<         u>         um*        um/mod     unchar     under+
unfilter              unloop    unstream             until      unused     value      variable
while      within     word      words      x.         xor        ~please
```

(Quicksort, continued.)

```
: OUNT    DUP @   1 CELLS UNDER+ ;
: PUSH            ( x anArray -- )
    DUP >R
        OUNT CELLS + !    ( )
    1 R> +!
;
: .NAMES          ( anArray -- )
    0 COL# !
    OUNT 0 ?DO          ( a-addr)
        I CELLS OVER + @ .WORD
    LOOP                DROP
;
```

We can now define SWORDS using the dataspace at HERE as a work array.

```
: SWORDS
    depth 0= ?? LAST      ( xt)
    ALIGN    0 HERE !
    begin
        ?DUP
    while
        dup >NAME HERE PUSH
        >PREVIOUS
    repeat                ( )
    HERE OUNT ['] CCOMPARE QSORT
    HERE .NAMES
;
```

I apologize for doing one of the worst things someone writing about a program can do—using undefined functions. However my purpose is to show a method. I don't know what the words will be in your system, and you will have to work that out yourself.

I hope that you have UNDER+ as a primitive in your system. A high-level definition if you don't is—

```
: UNDER+    rot + swap ;
```

UNDER+ should have been used in SkipLowers and Partition in the quicksort suite.

?? *thingy* is identical to IF *thingy* THEN. I don't know how to define it in your system.

The name OUNT is a pun which I'll leave for you to discover and groan over.

In This Forth dataspace and codespace are separate. "@" is for dataspace; "has" gives you what codespace has.

(Fast Forthward, continued from page 43.)

However, our experience with Forth vocabulary implementations is considerable. We should know about more alternatives by now. Some changes to consider are:

- Assuming that the current vocabulary should always be at the top of the search order, FIND could be altered to search the current vocabulary before searching the vocabularies in the vocabulary stack.
- If the current vocabulary should be at the top of the search order provisionally—such as only during compilation—then FIND could search the current vocabulary when the variable KERNEL-MANAGED-VOC is non-zero. The routines that invoke interpretation mode can set it to zero. The routines that invoke compilation mode can set it to a non-zero value.

Either of these options could help reduce the number of "cooks in the kitchen." The vocabulary stack could be left untouched by computer hands (so to speak). From a user-interface vantage, the vocabulary stack is better controlled when subject to explicit manipulation by the programmer only. It should not remain subject to implicit manipulation due to vocabulary operations buried inside a sprinkling of Forth kernel words.

We should try more varied resources for search-order control to see how they feel. Name-space management might be handled by scoping mechanisms more aligned with traditional programming languages.

Note that all these old and any new vocabulary-oriented tools seek the same thing. They seek to fine-tune the behavior of FIND's search loop according to various input-stream processing contexts. We should identify more precisely what those contexts are and why we need to cater to each one differently. Better solutions often come to us when we develop a complete understanding of the problem that we want to solve.

To Be Continued...

In summary, ineffective routine and user interfaces can be annoying to programmers and users. Defects are certain to reduce programmer productivity. Accordingly, the more conversant we become with interface design principles, the better off we will be. Besides, it will help us secure Forth's reputation as a rapid development tool.

Code refinement is often a matter of routine-to-routine interface refinement. Dismissing interfacing concerns merely as a religious debate to be carried on by the GUI and the non-GUI camps is therefore inappropriate.

The problem of obtaining finely tunable behavior from one routine so that it can serve well in several different contexts dogs us in many ways—particularly when a loop is responsible for the apparent inflexibility of a routine. I'll have more to say about the issue of coaxing flexibility out of routines that rely on loops in the next installment of this (growing) series.

Meanwhile, are there more Forth interface issues that you know about and that should be brought out into the open? Do you have any ideas about displacing vocabularies? I'd be interested in hearing your answers to these questions.

HARVARD SOFTWORKS

NUMBER ONE IN FORTH INNOVATION

(513) 748-0390 P.O. Box 69, Springboro, OH 45066

Just how good is HS/FORTH? Well, it's already good enough to control mile long irrigation arms to water the nations crops, good enough to control orbiting shuttle experiments, good enough to analyze the nation's blood supply and to control the telephone switching systems. It monitors pollution (nuclear and conventional) and simulates growth and decline of populations. It's good enough to simulate and control giant diesel generator engines and super cooled magnet arrays for particle accelerators. In the army and in the navy, at small clinics and large hospitals, even in the National Archives, HS/FORTH helps control equipment and manage data. It's good enough to control leading edge kinetic art, and even run light shows in New York's Metropolitan Museum of Art. Good enough to form the foundation of several of today's most innovative games (educational and just for fun), especially those with animation and mini-movies. If you've been zapping Romulans, governing nations, airports or train stations, or just learning to type - you may have been using HS/FORTH.

Our customers come from all walks of life. Doctors, lawyers and Indian Chiefs, astronomers and physicists, professional programmers and dedicated amateurs, students and retirees, engineers and hobbyists, soldiers and environmentalists, churches and social clubs. HS/FORTH was easy enough for all to learn, powerful enough to provide solutions, compact enough to fit on increasingly crowded disks. Give us a chance to help you too!

You can run HS/FORTH under DOS or **Microsoft Windows** in text and/or graphics windows with various icons and pif files for each. What I really like is cranking up the font size so I can still see the characters no matter how late it is. Now that's useful. There are few limits to program size since large programs simply grow into additional segments or even out onto disk. The Tools & Toys disk includes a complete mouse interface with menu support in both text and graphics modes. With HS/FORTH, one .EXE file and a collection of text files are all that you ever need. Since HS/FORTH compiles to executable code faster than most languages link, there is no need for wasteful, confusing intermediate file clutter.

HS/FORTH runs under MSDOS or PCDOS, or from ROM. Each level includes **all** features of lower ones. Level upgrades: $25. plus price difference between levels. Source code is in ordinary ASCII text files.

HS/FORTH supports megabyte and larger programs & data, and runs as fast as 64k limited Forths, even without automatic optimization -- which accelerates to near assembler language speed. Optimizer, assembler, and tools can load transiently. Resize segments, redefine words, eliminate headers without recompiling. Compile 79 and 83 Standard plus F83 programs.

PERSONAL LEVEL **$299.**
Fast direct to video memory text & scaled/clipped/windowed graphics in bit blit windows, mono, cga, ega, vga, all ellipsoids, splines, bezier curves, arcs, turtles; lightning fast pattern drawing even with irregular boundaries; powerful parsing, formatting, file and device I/O; DOS shells; interrupt handlers; call high level Forth from interrupts; single step trace, decompiler; music; compile 40,000 lines per minute, stacks; file search paths; format to strings. software floating point, trig, transcendental, 18 digit integer & scaled integer math; vars: A B * IS C compiles to 4 words, 1..4 dimension var arrays; **automatic optimizer delivers machine code speed.**

PROFESSIONAL LEVEL **$399.**
hardware floating point - data structures for all data types from simple thru complex 4D var arrays - operations complete thru complex hyperbolics; turnkey, seal; interactive dynamic linker for foreign subroutine libraries; round robin & interrupt driven multitaskers; dynamic string manager; file blocks, sector mapped blocks; x86&7 assemblers.

PRODUCTION LEVEL **$499.**
Metacompiler: DOS/ROM/direct/indirect; threaded systems start at 200 bytes, Forth cores from 2 kbytes; C data structures & struct+ compiler; MetaGraphics TurboWindow-C library, 200 graphic/window functions, PostScript style line attributes & fonts, viewports.

ONLINE GLOSSARY **$ 45.**

PROFESSIONAL and PRODUCTION LEVEL EXTENSIONS:

FOOPS+ with multiple inheritance **$ 79.**
TOOLS & TOYS DISK **$ 79.**
286FORTH or **386FORTH** **$299.**
16 Megabyte physical address space or gigabyte virtual for programs and data; DOS & BIOS fully and freely available; 32 bit address/operand range with 386.
ROMULUS HS/FORTH from ROM **$ 99.**

Shipping/system: US: $9. Canada: $21. foreign: $49. We accept MC, VISA, & AmEx

Understanding F83 Vocabulary Usage

Byron Nilsen
Oakland, California

The concept of "vocabularies" is a unique feature of the Forth language. We may ignore them at first, since a typical computer such as F83 defines the most popular words in the FORTH vocabulary, which is always available after the compiler is loaded. But eventually we will want to access words defined in other vocabularies, and when this need arises some confusion may follow.

Vocabularies provide a means to isolate groups of definitions so as to avoid conflicts that can occur if two or more definitions (in different parts of the program and dealing with different issues) are given the same name. They also admit a natural and easy way to preserve order in large applications programs, and can reduce compilation time by allowing the compiler to forego searching through groups of definitions that do not immediately pertain. Thoughtfully grouping definitions into various vocabularies with meaningful names can greatly enhance the interactive readability and maintainability of the program by other people—an important, but too-often-neglected consideration.

It is worthwhile to be aware of these behaviors and to observe a few simple rules...

Another valuable use of vocabularies is in creating "turnkey" applications wherein it is not desirable to expose the entire Forth lexicon. F83 supports the word SEAL which effectively "hides" all but one designated vocabulary. This is an easy way to protect a program from accidental or malicious tampering without having to rewrite all the keyboard-handling routines.

Or perhaps the programmer(s) might wish to keep new definitions in small "semi-private" vocabularies while a program is being developed. What Forth programmer hasn't had occasion to wonder exactly how the name of a word was spelled a few days earlier, when referencing it in a later definition? It's much easier to find (using WORDS) in a fairly small vocabulary. This is especially useful when several programmers are collaborating on a project.

The Forth-83 Standard does not prescribe how vocabularies are to be accessed, other than that the CONTEXT vocabulary shall be searched for pre-defined words, and the CURRENT vocabulary shall receive new definitions. F83 supports a search-order list of up to eight vocabularies, with that pointed to by CONTEXT being the first. There is no limit to how many vocabularies may exist, but only eight at a time can be in the search order. The CURRENT vocabulary is not searched unless it is also included in the CONTEXT search order.

As released by Laxen and Perry, F83 is comprised of these nine vocabularies:

ROOT	contains a small number of vocabulary-related words such as ONLY, ALSO, FORTH, etc. It will always be last in the search order.
BUG	holds "low-level" definitions used by the high-level words DEBUG and SEE. (Here, "high level" means "defined in the FORTH vocabulary.)
EDITOR	contains all the words unique to the screen-editing functions.
ASSEMBLER	contains the 8086 assembly definitions for CODE and LABEL.
FORTH	has all of the Forth-83 Standard definitions, plus the large collection of extension words provided by the authors of F83.
DOS	contains the definitions needed to interface with MS(PC)-DOS for disk and console access.
HIDDEN	contains some keyboard, display, and de-bug/decompile routines.
SHADOW	holds definitions that deal with shadow screens.
USER	holds definitions pertaining to multi-tasking.

New vocabularies are created by the defining word VOCABULARY. The header is identical to that of other definitions. The code field points into the parameter field of VOCABULARY to find its run-time action (CONTEXT!). The parameter field contains four thread pointers for dictionary hashing, and a link to the most recently created

vocabulary. The variable VOC-LINK points to the link cell in the parameter field of the most recently created vocabulary.

A multi-vocabulary search order is created by a series of words such as this:

FORTH	Ensure that FORTH is the CONTEXT vocabulary...
DEFINTIONS	...then make it the CURRENT vocabulary, too.
VOCABULARY NEWVOC	Create a new vocabulary within the FORTH vocabulary.
ONLY	Clear the context search order, leaving only ROOT.
FORTH	Make FORTH the context, with ROOT second-in-list.
ALSO	FORTH is CONTEXT and also second in search order.
DOS	DOS is now the context. FORTH is second-in-list.
ALSO	Replicate DOS as both CONTEXT and second-in-list.
NEWVOC	NEWVOC is now the context vocabulary. DOS is second.
ALSO	Replicate NEWVOC as the second vocabulary to be searched.
DEFINITIONS	Copy CONTEXT (NEWVOC) to CURRENT.

This produces an "active" vocabulary search order of:

Context: NEWVOC NEWVOC DOS FORTH ROOT
(This is the search order.)
Current: NEWVOC
(New definitions go here.)

(Why NEWVOC appears twice in the search order will be discussed below.)

The word ALSO replicates CONTEXT, pushing it down into the search-order list to the limit of eight. Thereafter, the vocabulary just above ROOT is removed from the list while CONTEXT is being replicated as the second element. The word PREVIOUS shifts the second vocabulary up into CONTEXT and shortens the search-order list by one. ORDER displays the search order, beginning with CONTEXT, and also shows the CURRENT vocabulary. VOCS displays the names of all vocabularies, whether or not they are part of the search order, in reverse order of creation (i.e., newest first). ONLY empties the search-order list, leaving ROOT.

The search order can be altered using "embedded directives" within a colon definition, by switching out of compile mode with the left-bracket ([) and naming the desired vocabulary. The named vocabulary is moved to CONTEXT, discarding what was there. Compilation continues at the right-bracket (]).

The programmer should be aware that, while F83 has no intrinsic mechanism to automatically "manage" the entire search order, several words do alter CONTEXT. This is usually done in such a way as to go unnoticed, but there are some situations that can be disruptive. Consider the following:

EDIT replicates CONTEXT (using ALSO) and makes EDITOR the CONTEXT vocabulary. DONE finishes the edit, updates the disk buffer, and restores the original context by executing PREVIOUS. Attempting to interactively compile a word while editing is possible, but may cause headaches. It is generally wise to eschew such activities when EDIT is active.

CODE saves the CONTEXT vocabulary in a variable, then invokes ASSEMBLER. The original context will be restored by END-CODE. *But*—an error will occur if the new code definition calls for a variable or constant that was defined in the original CONTEXT vocabulary and *if a second reference to that vocabulary is not in the search order!* The word will not be found! This potential problem could, perhaps, be avoided by rewriting CODE and END-CODE to mimic EDIT and DONE. It's better just to have an extra reference elsewhere in the list.

LABEL makes ASSEMBLER the CONTEXT vocabulary *but does not save the original context.* Because label definitions do not terminate with END-CODE, ASSEMBLER remains as the CONTEXT vocabulary. As with CODE, LABEL could be redefined (by adding ALSO) to at least postpone the loss of a vocabulary from the search order. But then, repeated label definitions would soon push everything except ASSEMBLER and ROOT out of the list! This implies that label definitions must be terminated with a word similar to DONE. Not necessary! Here again, no problem exists so long as CONTEXT is replicated in the list. (Restoring the original search order by executing PREVIOUS and ALSO after compiling a label definition is good practice, but should not really be necessary.)

The colon (:) copies CURRENT into CONTEXT. This is done so as to consistently restore CONTEXT in the event that it has been altered by a LABEL or embedded directive. There is a very good possibility that the programmer wants the CURRENT vocabulary and the CONTEXT vocabulary to be the same anyway. Thus, when a colon definition follows a labeled one, everything returns to normal. *But,* if an attempt is made to "temporarily" include a vocabulary by making it the CONTEXT before beginning to compile a colon definition, *it will be discarded* by execution of the colon! (This is where embedded directives are useful.) Being always obliged to have CONTEXT and CURRENT the same might annoy the programmer who seeks maximum compilation speed when all constituents of new words to be compiled into one vocabulary just happen to be defined in another. "Ideally," the vocabulary holding those definitions should be first in the search order. If this is really important enough, the colon itself may be redefined, omitting the portion that copies CURRENT into CONTEXT.

Compiler directives embedded in a colon definition (e.g., ... [DOS] ...) should pose no problem, so long as the extra context reference is present. *However,* a problem can occur if a definition holds an embedded directive to change the *number base* (e.g., ... [HEX] ...) and the programmer—noting that CONTEXT is always restored to what it was before (same as CURRENT)—assumes that all such embedded directives are in effect

FIG
MAIL ORDER FORM

HOW TO USE THIS FORM: Please enter your order on the back page of this form and send with your payment to the Forth Interest Group. All items have one price and a weight marked with a # sign. Enter weight on order form and calculate shipping based on location and delivery method.

"Were Sure You Wanted To Know..."

***Forth Dimensions*, Article Reference** 151 - $4 0#
★ An index of Forth articles, by keyword, from *Forth Dimensions* Volumes 1–14 (1978–93).

FORML, Article Reference 152 - $4 0#
★ An index of Forth articles by keyword, author, and date from the FORML Conference Proceedings (1980–93).

FORTH DIMENSIONS BACK VOLUMES

A volume consists of the six issues from the volume year (May–April)

Volume 1 Forth Dimensions (1979–80) 101 - $15 1#
Introduction to FIG, threaded code, TO variables, fig-Forth.

Volume 6 Forth Dimensions (1984–85) 106 - $15 2#
Interactive editors, anonymous variables, list handling, integer solutions, control structures, debugging techniques, recursion, semaphores, arrays, simple I/O words, Quicksort, high-level packet communications, China FORML.

Volume 7 Forth Dimensions (1985–86) 107 - $20 2#
Generic sort, Forth spreadsheet, control structures, pseudo-interrupts, number editing, Atari Forth, pretty printing, code modules, universal stack word, polynomial evaluation, F83 strings.

Volume 8 Forth Dimensions (1986–87) 108 - $20 2#
Interrupt-driven serial input, data-base functions, TI 99/4A, XMODEM, on-line documentation, dual CFAs, random numbers, arrays, file query, Batcher's sort, screenless Forth, classes in Forth, Bresenham line-drawing algorithm, unsigned division, DOS file I/O.

Volume 9 Forth Dimensions (1987–88) 109 - $20 2#
Fractal landscapes, stack error checking, perpetual date routines, headless compiler, execution security, ANS-Forth meeting, computer-aided instruction, local variables, transcendental functions, education, relocatable Forth for 68000.

Volume 10 Forth Dimensions (1988–89) 110 - $20 2#
dBase file access, string handling, local variables, data structures, object-oriented Forth, linear automata, stand-alone applications, 8250 drivers, serial data compression.

Volume 11 Forth Dimensions (1989–90) 111 - $20 2#
Local variables, graphic filling algorithms, 80286 extended memory, expert systems, quaternion rotation calculation, multiprocessor Forth, double-entry bookkeeping, binary table search, phase-angle differential analyzer, sort contest.

Volume 12 Forth Dimensions (1990–91) 112 - $20 2#
Floored division, stack variables, embedded control, Atari Forth, optimizing compiler, dynamic memory allocation, smart

FORML CONFERENCE PROCEEDINGS

FORML (Forth Modification Laboratory) is an educational forum for sharing and discussing new or unproven proposals intended to benefit Forth, and is an educational forum for discussion of the technical aspects of applications in Forth. Proceedings are a compilation of the papers and abstracts presented at the annual conference. FORML is part of the Forth Interest Group.

1981 FORML PROCEEDINGS 311 - $45 4#
CODE-less Forth machine, quadruple-precision arithmetic, overlays, executable vocabulary stack, data typing in Forth, vectored data structures, using Forth in a classroom, pyramid files, BASIC, LOGO, automatic cueing language for multimedia, NEXOS—a ROM-based multitasking operating system. *655 pgs*

1982 FORML PROCEEDINGS 312 - $30 4#
Rockwell Forth processor, virtual execution, 32-bit Forth, ONLY for vocabularies, non-IMMEDIATE looping words, number-input wordset, I/O vectoring, recursive data structures, programmable-logic compiler. *295 pgs*

1983 FORML PROCEEDINGS 313 - $30 2#
Non-Von Neuman machines, Forth instruction set, Chinese Forth, F83, compiler & interpreter co-routines, log & exponential function, rational arithmetic, transcendental functions in variable-precision Forth, portable file-system interface, Forth coding conventions, expert systems. *352 pgs*

1984 FORML PROCEEDINGS 314 - $30 2#
Forth expert systems, consequent-reasoning inference engine, Zen floating point, portable graphics wordset, 32-bit Forth, HP71B Forth, NEON—object-oriented programming, decompiler design, arrays and stack variables. *378 pgs*

1986 FORML PROCEEDINGS 316 - $30 2#
Threading techniques, Prolog, VLSI Forth microprocessor, natural-language interface, expert system shell, inference engine, multiple-inheritance system, automatic programming environment. *323 pgs*

1987 FORML PROCEEDINGS 317 - $40 3#
Includes papers from '87 euroFORML Conference. 32-bit Forth, neural networks, control structures, AI, optimizing compilers, hypertext, field and record structures, CAD command language, object-oriented lists, trainable neural nets, expert systems. *463 pgs*
Last 5

1988 FORML PROCEEDINGS 318 - $40 2#
Includes 1988 Australian FORML, Human interfaces, simple robotics kernel, MODUL Forth, parallel processing, programmable controllers, Prolog, simulations, language topics, hardware, Wil's workings & Ting's philosophy, Forth hardware applications, ANS Forth session, future of Forth in AI applications. *310 pgs*

1989 FORML PROCEEDINGS 319 - $40 3#
Includes papers from '89 euroFORML. Pascal to Forth, extensible optimizer for compiling, 3D measurement with object-oriented Forth, CRC polynomials, F-PC, Harris C cross-compiler, modular approach to robotic control, RTX recompiler for on-line maintenance, modules, trainable neural nets. *433 pgs*

1991 FORML PROCEEDINGS 321 - $50 3#
Includes 1991 FORML (Asilomar), euroFORML '91 (Czechoslovakia) and 1991 China FORML (Shanghai). differential file comparison, LINDA on a simulated network, QS2: RISCing it all, A threaded microprogram machine, Forth in networking, Forth in the Soviet Union, FOSM: A FOrth String Matcher, VGA Graphics and 3-D animation, Forth and TSR, Forth CAE system, applying Forth to electric discharge machining, MCS96-FORTH single chip computer. *500 pgs*
Last 5

1992 FORML PROCEEDINGS 322 - $40 2#
Object oriented Forth bases on classes rather than prototypes, color vision sizing processor, virtual file systems, transparent target development, Signal processing pattern classification, optimization in low level Forth, local variables, embedded Forth, auto display of digital images, graphics package for F-PC, B-tree in Forth *200 pgs*

★ These are your most up-to-date indexes for back issues of *Forth Dimensions* and the FORML proceedings.

Fax your orders: 510-535-1295

BOOKS ABOUT FORTH

ALL ABOUT FORTH, 3rd ed., June 1990, Glen B. Haydon 201 - $90 4#
Annotated glossary of most Forth words in common usage, including Forth-79, Forth-83, F-PC, MVP-Forth. Implementation examples in high-level Forth and/or 8086/88 assembler. Useful commentary given for each entry. *504 pgs*

eFORTH IMPLEMENTATION GUIDE, C.H. Ting 215 - $25 1#
eForth is the name of a Forth model designed to be portable to a large number of the newer, more powerful processors available now and becoming available in the near future. *54 pgs* (w/disk)

Embedded Controller FORTH, 8051, William H. Payne 216 - $65 2#
Describes the implementation of an 8051 version of Forth. More than half of this book contains source listings (w/disks C050) *511 pgs*

F83 SOURCE, Henry Laxen & Michael Perry 217 - $20 2#
A complete listing of F83, including source and shadow screens. Includes introduction on getting started. *208 pgs*

FORTH: A TEXT AND REFERENCE 219 - $31 2#
Mahlon G. Kelly & Nicholas Spies
A textbook approach to Forth, with comprehensive references to MMS-FORTH and the '79 and '83 Forth standards. *487 pgs*

THE FIRST COURSE, C.H. Ting 223 - $25 1#
This tutorial's goal is to expose you to the very minimum set of Forth instructions you need to use Forth to solve practical problems in the shortest possible time. "... This tutorial was developed to complement *The Forth Course* which skims too fast on the elementary Forth instructions and dives too quickly in the advanced topics in a upper level college microcomputer laboratory ..." A running F-PC Forth system would be very useful. *44 pgs*

THE FORTH COURSE, Richard E. Haskell 225 - $25 1#
This set of 11 lessons, called *The Forth Course*, is designed to make it easy for you to learn Forth. The material was developed over several years of teaching Forth as part of a senior/graduate course in design of embedded software computer systems at Oakland University in Rochester, Michigan. *156 pgs* (w/disk)

FORTH ENCYCLOPEDIA, Mitch Derick & Linda Baker 220 - $30 2#
A detailed look at each fig-Forth instruction. *327 pgs*

FORTH NOTEBOOK, Dr. C.H. Ting 232 - $25 2#
Good examples and applications. Great learning aid. poly-FORTH is the dialect used. Some conversion advice is included. Code is well documented. 286 pgs

FORTH NOTEBOOK II, Dr. C.H. Ting 232a - $25 2#
Collection of research papers on various topics, such as image processing, parallel processing, and miscellaneous applications. *237 pgs*

FORTH: The New Model, Jack Woehr 233 - $45 2#
This book teaches Forth and the proposed new standard from the perspective of a Technical Committee member. You will find it especially helpful if you are: An experienced Forth programmer who wishes to become familiar with the draft-proposed standard for Forth, a Forth programmer who needs to know how to convert existing programs to the new proposed standard, a programmer, experienced in other languages, who is using Forth for an embedded control project, or a beginning Forth programmer who wishes to learn the language. *315 pgs, w/disk*

F-PC USERS MANUAL (2nd ed., V3.5) 350 - $20 1#
Users manual to the public-domain Forth system optimized for IBM PC/XT/AT computers. A fat, fast system with many tools. *143 pgs*

F-PC TECHNICAL REFERENCE MANUAL 351 - $30 2#
A must if you need to know the inner workings of F-PC. *269 pgs*

INSIDE F-83, Dr. C.H. Ting 235 - $25 2#
Invaluable for those using F-83. *226 pgs*

LIBRARY OF FORTH ROUTINES AND UTILITIES,
James D. Terry 237 - $23 2#
Comprehensive collection of professional quality computer code for Forth; offers routines that can be put to use in almost any Forth application, including expert systems and natural-language interfaces. *374 pgs*

OBJECT-ORIENTED FORTH, Dick Pountain 242 - $35 1#
Implementation of data structures. First book to make object-oriented programming available to users of even very small home computers. *118 pgs*

SEEING FORTH, Jack Woehr 243 - $25 1#
"... I would like to share a few observations on Forth and computer science. That is the purpose of this monograph. It is offered in the hope that it will broaden slightly the streams of Forth literature ..." *95 pgs*

SCIENTIFIC FORTH, Julian V. Noble 250 - $50 2#
Scientific Forth extends the Forth kernel in the direction of scientific problem solving. It illustrates advanced Forth programming techniques with non-trivial applications: computer algebra, roots of equations, differential equations, function minimization, functional representation of data (FFT, polynomials), linear equations and matrices, numerical integration/Monte Carlo methods, high-speed real and complex floating-point arithmetic. *300 pgs* (Includes disk with programs and several utilities), **IBM**

STACK COMPUTERS, THE NEW WAVE 244 - $62 2#
Philip J. Koopman, Jr. (hardcover only)
Presents an alternative to Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC) by showing the strengths and weaknesses of stack machines.

STARTING FORTH (2nd ed.), Leo Brodie 245 - $29 2#
In this edition of *Starting Forth*—the most popular and complete introduction to Forth—syntax has been expanded to include the Forth-83 Standard. *346 pgs*

WRITE YOUR OWN PROGRAMMING LANGUAGE USING C++,
Norman Smith 270 - $15 1#
This book is about an application language. More specifically, it is about how to write your own custom application language. The book contains the tools necessary to begin the process and a complete sample language implementation. [Guess what language!] Includes disk with complete source. *108 pgs*

ACM - SIGFORTH

The ACM SIGForth Newsletter is published quarterly by the Association of Computing Machinery, Inc. SIGForth's focus is on the development and refinement of concepts, methods, and techniques needed by Forth professionals.

Volume 1 #1-#4 911 - $24 2#
F-PC, glossary utility, euroForth, SIGForth '89 Workshop summary (real-time software engineering), Intel 80x8x. Metacompiler in cmForth, Forth exception handler, string case statement for UF/Forth. 1802 simulator, tutorial on multiple threaded vocabularies. Stack frames, duals: an alternative to variables, PocketForth.

Volume 2 #1-#4 912 - $24 2#
ACM SIGForth Industry Survey, abstracts 1990 Rochester conf., RTX-2000. BNF Parser, abstracts 1990 Rochester conf., F-PC Teach. Tethered Forth model, abstracts 1990 SIGForth conf. Target-meta-cross-: an engineer's viewpoint, single-instruction computer.

Volume 3, #1-#4 913 - $24 2#
Co-routines and recursion for tree balancing, convenient number handling. Postscript Issue, What is Postscript?, Forth in Postscript, Review: PS-Tutor.

For faster service, fax your orders: 510-535-1295

DISKS: Contributions from the Forth Community

The "Contributions from the Forth Community" disk library contains author-submitted donations, generally including source, for a variety of computers & disk formats. Each file is determined by the author as public domain, shareware, or use with some restrictions. This library does not contain "For Sale" applications. *To submit your own contributions, send them to the FIG Publications Committee.*

Count any number of disks as equal to 1 #.

FLOAT4th.BLK V1.4 Robert L. Smith C001 - $8
Software floating-point for fig-, poly-, 79-Std., 83-Std. Forths. IEEE short 32-bit, four standard functions, square root and log.
★★★ **IBM, 190Kb, F83**

Games in Forth C002 - $6
Misc. games, Go, TETRA, Life... Source.
★ **IBM, 760Kb**

A Forth Spreadsheet, Craig Lindley C003 - $6
This model spreadsheet first appeared in *Forth Dimensions* VII/1,2. Those issues contain docs & source.
★ **IBM, 100Kb**

Automatic Structure Charts, Kim Harris C004 - $8
Tools for analysis of large Forth programs, first presented at FORML conference. Full source; docs incl. in 1985 FORML Proceedings.
★★ **IBM, 114Kb**

A Simple Inference Engine, Martin Tracy C005 - $8
Based on inf. engine in Winston & Horn's book on LISP, takes you from pattern variables to complete unification algorithm, with running commentary on Forth philosophy & style. Incl. source.
★★ **IBM, 162 Kb**

The Math Box, Nathaniel Grossman C006 - $10
Routines by foremost math author in Forth. Extended double-precision arithmetic, complete 32-bit fixed-point math, & auto-ranging text. Incl. graphics. Utilities for rapid polynomial evaluation, continued fractions & Monte Carlo factorization. Incl. source & docs.
★★ **IBM, 118 Kb**

AstroForth & AstroOKO Demos, I.R. Agumirsian C007 - $6
AstroForth is the 83-Std. Russian version of Forth. Incl. window interface, full-screen editor, dynamic assembler & a great demo. AstroOKO, an astronavigation system in AstroForth, calculates sky position of several objects from different earth positions. Demos only.
★ **IBM, 700 Kb**

Forth List Handler, Martin Tracy C008 - $8
List primitives extend Forth to provide a flexible, high-speed environment for AI. Incl. ELISA and Winston & Horn's micro-LISP as examples. Incl. source & docs.
★★ **IBM, 170 Kb**

8051 Embedded Forth, William Payne C050 - $20
8051 ROMmable Forth operating system. 8086-to-8051 target compiler. Incl. source. Docs are in the book *Embedded Controller Forth for the 8051 Family.* Included with item #216
★★★ **IBM HD, 4.3 Mb**

68HC11 Collection C060 - $16
Collection of Forths, tools and floating point routines for the 68HC11 controller.
★★★ **IBM HD, 2.5 Mb**

F83 V2.01, Mike Perry & Henry Laxen C100 - $20
The newest version, ported to a variety of machines. Editor, assembler, decompiler, metacompiler. Source and shadow screens. Manual available separately (items 217 & 235). Base for other F83 applications.
★ **IBM, 83, 490 Kb**

F-PC V3.5615 & **TCOM**, Tom Zimmer C200 - $30
A full Forth system with pull-down menus, sequential files, editor, forward assembler, metacompiler, floating point. Complete source and help files. Manual for V3.5 available separately (items 350 & 351). Base for other F-PC applications.
★ **IBM HD, 83, 3.5Mb**

F-PC TEACH V3.5, Lessons 0–7 Jack Brown C201 - $8
Forth classroom on disk. First seven lessons on learning Forth, from Jack Brown of B.C. Institute of Technology.
★ **IBM HD, F-PC, 790 Kb**

VP-Planner Float for F-PC, V1.01 Jack Brown C202 - $8
Software floating-point engine behind the VP-Planner spreadsheet. 80-bit (temporary-real) routines with transcendental functions, number I/O support, vectors to support numeric co-processor overlay & user NAN checking.
★★ **IBM, F-PC, 350 Kb**

F-PC Graphics V4.6, Mark Smiley C203 - $10
The latest versions of new graphics routines, including CGA, EGA, and VGA suppport, with numerous improvements over earlier versions created or supported by Mark Smiley.
★★ **IBM HD, F-PC, 605 Kb**

PocketForth V6.1, Chris Heilman C300 - $12
Smallest complete Forth for the Mac. Access to all Mac functions, events, files, graphics, floating point, macros, create standalone applications and DAs. Based on fig & *Starting Forth.* Incl. source and manual.
★ **MAC, 640 Kb,** System 7.01 Compatible.

Kevo V0.9b6, Antero Taivalsaari C360 - $10
Complete Forth-like object Forth for the Mac. Object-Prototype access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Kernel source included, extensive demo files, manual.
★★★ **MAC, 650 Kb,** System 7.01 Compatible.

Yerkes Forth V3.66 C350 - $20
Complete object-oriented Forth for the Mac. Object access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Incl. source, tutorial, assembler & manual.
NEW VERSION
★★ **MAC, 2.4Mb,** System 7.1 Compatible.

Pygmy V1.4, Frank Sergeant C500 - $20
A lean, fast Forth with full source code. Incl. full-screen editor, assembler and metacompiler. Up to 15 files open at a time.
★★ **IBM, 320 Kb**

KForth, Guy Kelly C600 - $20
A full Forth system with windows, mouse, drawing and modem packages. Incl. source & docs.
★★ **IBM, 83, 2.5 Mb**

Mops V2.3, Michael Hore C710 - $20
Close cousin to Yerkes and Neon. Very fast, compiles subroutine-threaded & native code. Object oriented. Uses F-P co-processor if present. Full access to Mac toolbox & system. Supports System 7 (e.g., AppleEvents). Incl. assembler, manual & source.
★★ **MAC, 3 Mb,** System 7.1 Compatible

BBL & Abundance, Roedy Green C800 - $30
BBL public-domain, 32-bit Forth with extensive support of DOS, meticulously optimized for execution speed. Abundance is a public-domain database language written in BBL. Incl. source & docs.
★★★ **IBM HD, 13.8 Mb, hard disk required**

Version-Replacement Policy

Return the old version with the FIG labels and get a new version replacement for 1/2 the current version price.

MISCELLANEOUS

T-SHIRT "May the Forth Be With You" 601 - $12 1#
(Specify size: Small, Medium, Large, X-Large on order form)
white design on a dark blue shirt or green design on tan shirt.

Last 10

POSTER (*Oct., 1980 BYTE* cover) 602 - $5 1#

FORTH-83 HANDY REFERENCE CARD 683 - free

FORTH-83 STANDARD 305 - $15 1#
Authoritative description of Forth-83 Standard. For reference, not instruction. *83 pgs*

BIBLIOGRAPHY OF FORTH REFERENCES 340 - $18 2#
(3rd ed., January 1987)
Over 1900 references to Forth articles throughout computer literature. *104pgs*

MORE ON FORTH ENGINES

Volume 10 January 1989 810 - $15 1#
RTX reprints from 1988 Rochester Forth conference, object-oriented cmForth, lesser Forth engines. *87 pgs*

Volume 11 July 1989 811 - $15 1#
RTX supplement to *Footsteps in an Empty Valley*, SC32, 32-bit Forth engine, RTX interrupts utility. *93 pgs*

Volume 12 April 1990 812 - $15 1#
ShBoom Chip architecture and instructions, neural computing module NCM3232, pigForth, binary radix sort on 80286, 68010, and RTX2000. *87 pgs*

Volume 13 October 1990 813 - $15 1#
PALs of the RTX2000 Mini-BEE, EBForth, AZForth, RTX2101, 8086 eForth, 8051 eForth. *107 pgs*

Volume 14 814 - $15 1#
RTX Pocket-Scope, eForth for muP20, ShBoom, eForth for CP/M & Z80, XMODEM for eForth. *116 pgs*

Volume 15 815 - $15 1#
Moore: new CAD system for chip design, a portrait of the P20; Rible: QS1 Forth processor, QS2, RISCing it all; P20 eForth software simulator/debugger. *94 pgs*

Volume 16 816 - $15 1#
OK-CAD System, MuP20, eForth system words, 386 eForth, 80386 protected mode operation, FRP 1600 - 16-Bit real time processor. *104 pgs*

Volume 17 817 - $15 1#
P21 chip and specifications; Pic17C42; eForth for 68HC11, 8051, Transputer *128 pgs*

DR. DOBB'S JOURNAL back issues

Annual Forth issue, includes code for various Forth applications.

Sept. 1982, Sept. 1983, Sept. 1984 (3 issues) 425 - $10 1#

JFAR BACK ISSUES

Rochester, 1981	Standards Conference	751 - $25 2#
Rochester, 1989	Industrial Automation	759 - $25 2#
Rochester, 1990	Embedded Systems	760 - $30 2#
Rochester, 1991	Automated Systems	761 - $30 2#

Last 5

FORTH INTEREST GROUP

P.O. BOX 2154 OAKLAND, CALIFORNIA 94621 510-89-FORTH (510-893-6784) 510-535-1295 (FAX)

Name _____ Phone _____
Company _____ Fax _____
Street _____ eMail _____
City _____
State/Prov. _____ Zip _____
Country _____

U.S. Domestic Postage Rates	Surface*** $1.00/#	2 day Priority $1.50/#
International Postage Rates	Surface All /#	AIR MAIL 1-4 #s/#	AIR MAIL >4 #s/#
Canada, Mexico	$1.00	$2.00	$1.30
Other Western Hemisphere	$1.00	$3.25	$2.25
Europe	$1.00	$6.00	$4.50
Other International	$1.00	$8.00	$6.00

*Non-Post Office deliveries: Include Special Instructions.

Item #	Title	Qty.	Unit Price	Total	#

☐ CHECK ENCLOSED (Payable to: FIG)
☐ VISA/MasterCard Expiration Date _____
Card Number _____
Signature _____

Sub-Total		
10% Member Discount, Member # _____	( )	#s times rate
**Sales Tax on Sub-Total (CA only)		
Postage: Rate _____ x #s		
*Membership in the Forth Interest Group ☐New ☐Renewal $40/46/52		
MEMBERSHIP Total		

*MEMBERSHIP IN THE FORTH INTEREST GROUP

The Forth Interest Group (FIG) is a world-wide, non-profit, member-supported organization with over 1,500 members and 40 chapters. Your membership includes a subscription to the bi-monthly magazine *Forth Dimensions*. FIG also offers its members an on-line data base, a large selection of Forth literature and other services. Cost is $40 per year for U.S.A. & Canada surface; $46 Canada air mail; all other countries $52 per year. This fee includes $36 for *Forth Dimensions*. No sales tax, handling fee, or discount on membership.
When you join, your first issue will arrive in four to six weeks; subsequent issues will be mailed to you every other month as they are published—six issues in all. Your membership entitles you to a 10% discount on publications and functions of FIG. Dues are not deductible as a charitable contribution for U.S. federal income tax purposes, but may be deductible as a business expense.

MAIL ORDERS:
Forth Interest Group
P.O. Box 2154
Oakland, CA 94621
PHONE ORDERS:
510-89-FORTH Credit card orders, customer service. Hours: Mon–Fri, 9–5 p.m.

PAYMENT MUST ACCOMPANY ALL ORDERS

PRICES: All orders must be prepaid. Prices are subject to change without notice. Credit card orders will be sent and billed at current prices. Checks must be in U.S. dollars, drawn on a U.S. bank. A $10 charge will be added for returned checks.

POSTAGE:
All orders calculate postage as number of #s times selected postage rate.
Special handling available on request.

SHIPPING TIME:
Books in stock are shipped within seven days of receipt of the order.
***SURFACE DELIVERY:**
US: 10 days, Other: 30-60 days

**** CALIFORNIA SALES TAX BY COUNTY:**
7.75%: Del Norte, Fresno, Imperial, Inyo, Madera, Orange, Riverside, Sacramento, San Benito, Santa Barbara, San Bernardino, San Diego, and San Joaquin; **8.25%**: Alameda, Contra Costa, Los Angeles San Mateo, San Francisco, Santa Clara, and Santa Cruz; **7.25%**: other counties.

For faster service, fax your orders: 510-535-1295

only until completion of that definition! Also, attempting to continue compiling after a "crash" may be unsuccessful if the search order was altered by any of the above events.

Summary

Many Forth programmers will probably lead long and productive lives writing much useful code without ever noticing what might be happening to the CONTEXT vocabulary during compilation. It is, nevertheless, worthwhile to be aware of these behaviors and to observe a few simple rules:

- Always replicate the CONTEXT vocabulary with ALSO before beginning compilation—most certainly when CODE or LABEL definitions are planned, or if embedded, context-altering directives are expected. The compiler's dictionary-search routine (FIND) anticipates this replication and skips over a vocabulary if it is the same as that just previously scanned, thereby diminishing any compilation-speed penalty.
- Be wary of embedded compiler directives. Remember that anything done between the brackets remains in effect until explicitly changed again.
- Use caution if performing any non-editing operations while a screen edit is in progress. An extra reference to the original CONTEXT vocabulary can help to avoid some problems here, but a risk of trashing the edit does exist.
- Be content that CONTEXT and CURRENT will normally be the same vocabulary during colon compilation.

(Interactive Embedded Development, from page 11.)

systems, he admitted that the system his company had just bought for $8000 did not have the interactiveness described here. Our software cost about one-tenth as much, and our system used no special hardware.

This will be a good challenge for anyone doing embedded systems programming. I have used it as described here without problems for a couple of major projects. It's beautiful to be able to hit the backspace key, change just a digit or a word, and instantly try your code again, without leaving the editor, and *on* the target to boot! It also seems to improve your chances of finding and fixing bugs early in the game. The automated test system I mentioned earlier has been in daily use for nearly two years, and not a single bug has shown up in that time.

I hope this description was clear enough for many to pick it up and run with it. After all, crawling is not the Forth way.

Garth Wilson began programming in Fortran and assembly in college in 1982. Three types of BASIC and Forth were among the languages he later used for data acquisition and automated test equipment. He wrote the code for a flight-following computer in assembly. Much of his early programming was on a series of TI and HP hand-held programmables, which he used to facilitate a wide range of work. A friend told him a little about Forth in 1985, but it wasn't until 1989 that he picked up Brodie's book and started getting to know Forth. As a project to learn Forth, he wrote a cross-assembler and linker program. He enjoyed the language immensely, and was delighted to see development time plunge. Programming has been a part of his job since 1986. Now he is part owner of an aircraft communications company. He can be reached by phone at 310-695-7054 or by mail at 11123 Dicky Street, Whittier, California 90606.

Generation and Application of Random Numbers

[Article continues in next issue, including Listings Three, Four, and Five, and Figure Three.]

Dr. Everett F. Carter, Jr.
Monterey, California

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.
—John Von Neumann (1951)

1. Introduction

The famous statement of John von Neumann notwithstanding, the generation of random numbers with a computer is a major enterprise. In fact it is estimated that the world's computers generate ten *billion* random numbers *per second* (Cooper, 1989). What von Neumann meant, of course, is that it is not so easy to generate a random number with a deterministic algorithm. There are many subtle problems that can occur and various compromises have to be made in order to even pretend to generate random numbers with a computer. In this article we will explore the generation of random numbers and some important applications that use such numbers. We will find that there is no universal solution because different applications demand different properties of our numbers.

The Forth code that accompanies this article (Listings One through Five) is written using F-PC V3.56. It requires the tools files `DMULDIV.SEQ` for extra words that handle double-precision integers, and `FFLOAT.SEQ` for extra floating-point words. Both of these extra files are ordinarily distributed with the F-PC package. Listing Five, `FILEIO.SEQ`, is a collection of support words to make file I/O simpler.

2. Required Properties of a Good Generator

A good random number generator must reasonably represent a known probability distribution function (usually uniform over some finite domain). We do not want any built-in trends, biases, or periodicities. We usually do not want the value generated at a given time to be correlated in any way with previous values (see below for exceptions).

There are many ways to test the quality of a random number generator. The two most important quantitative ones are the χ^2 test and the lagged correlation.

The lagged correlation will reveal the relationship between the numbers at one time and at another; it also can reveal trends and periodicities. The correlation at lag τ is calculated by the formula,

$$\rho(\tau) = \frac{\sum_i x_i y_{i+\tau} - \frac{1}{n-\tau}\sum_i x_i \sum_i y_{i+\tau}}{\sqrt{\sum_i x_i^2 \; \sum_i y_{i+\tau}^2}} \qquad (1)$$

(all sums are over $n - \tau$ points in the n point data set). Usually this quantity is called the *cross-correlation* when x and y represent two different data sets. It is called the *auto-correlation* when x and y are the same data. In this application (and in the listing), we are interested in the auto-correlation.

An ideal random number generator will give an auto-correlation value of 1.0 for $\tau = 0$, and a value of 0.0 for any other value of τ. A significant peak at non-zero values of τ indicates that there is a periodicity at that time lag. If the correlation values slowly drop to zero as τ increases, then the numbers are not very independent of each other.

The χ^2 test is for measuring how well the presumed distribution (generally uniform for these canonical generators) is represented. This test works the following way. First divide up the whole interval that the random number will be within into a finite number of bins or class intervals (these intervals need not be of the same size, but for simplicity we will take them to be here). Then we count the number of random numbers observed within each interval and calculate the *expected* number of observations (when the intervals are uniform, this is just the number of random numbers used divided by the number of class intervals). Then we calculate the sum,

$$\chi^2 = \sum_{i=1}^{m} \frac{(observed_i - expected_i)^2}{expected_i} \qquad (2)$$

Note that m is the *number of class intervals*, not the number of data points. As long as none of the observed counts is wildly different from the expected values (in particular if none of them is zero), then the χ^2 value can estimate the probability that the observed data is a representative sample of the expected distribution (by

looking up the above value in a table of critical χ^2 values). For this test, small values are good values. Generally, small means less than the number of class intervals. Statistics tables either give the χ^2 value as a function of the *degrees of freedom* or give the *reduced* χ^2, which is the above quantity divided by the degrees of freedom. In this context the number of degrees of freedom is just the number of class intervals minus one (m–1).

Listing One, `STATS.SEQ`, implements the auto-correlation and the χ^2 tests. Both of the Forth words `cor` and `chi_2` read the data to be analyzed from a named file. I used the χ^2 test on a test set of 2000 double random numbers. These were created with `test_drand` with `drandgen` set to `r250d`. The result gave $\chi^2 = 21.34$ for 29 degrees of freedom. If we wanted to be 90% confident that we were sampling a uniform distribution, then the statistics tables say that we should get a value less than 39.09. Since we *did* get a smaller value, then we accept the hypothesis that a uniform distribution is being sampled (with 90% confidence).

Beyond these tests, my favorites are very powerful *qualitative* measures. One is to plot pairs of random numbers in a scatter plot, like Figure One. With this kind of plot, clumps of number, gaps, and patterns can easily be seen. The second test is the random walk test. To generate a random walk in two dimensions, one divides the range of the random number generator into four equal intervals. Then generate a number. If the value falls in the first interval, increment the X value. If the number falls in the second, increment the Y value. A number in the third interval means decrement X. And finally, a number in the fourth interval means decrement the Y value. For an ordinary random walk, the mean squared distance from the origin is linearly proportional to the amount of time for the walk. So, generate t steps of a random walk for n walks, calculate the mean squared distance reached (averaged over the n walks), and plot this distance versus time. The word `walk_test` in Listing Two (`RANTST.SEQ`) will generate a set of random walks and write the mean square distances (for X and Y separately) to a named file. A plot for several values of t and distance *should* be roughly linear, like Figure Two-a; if it bends over like in Figure Two-b or shows periodicities, then there is a problem.

3. Linear Congruential Generators

One of the most popular methods for generating random numbers is the *linear congruential generator.* These generators use a method similar to the folding schemes in chaotic maps. The general formula is,

$$I_k = (aI_{k-1}+c) mod\ m \qquad (3)$$

The values a, c, and m are pre-selected constants. a is known as the multiplier, c is the increment, and m is the modulus. The quality of the generator is strongly dependent upon the choice of these constants (a significant part of Knuth's chapter on random numbers is dedicated to this topic). The method is appealing however, because once a good set of the parameters is found, it is very easy to program. One fairly obvious goal is to make the period (the time before the generator repeats the sequence) long; this implies that m should be as large as possible. This means that 16-bit random numbers generated by this method have *at most* a period of 65,536, which is not nearly long enough for serious applications.

The choice of $a = 1277$, $c = 0$, $m = 131072$ looks okay for a while but eventually has problems. Figure One-b is an XY plot for this generator for 2000 pairs; one sees linear bands emerging from the plot. The random walk plot (Figure Two-b) shows that after a while the slope changes. This generator is implemented in the word `lcm_bad` in Listing Three (`RANDS.SEQ`).

The choice of $a = 16807$, $c = 0$, $m = 2147483647$ is a very good set of parameters for this generator. These parameters were published by Park & Miller (1988). This generator often is known as the *minimal standard* random number generator; it is often (but not always) the generator used for the built-in random number function in compilers and other software packages. This generator is implemented in the word `lcm_rand` in Listing Three.

4. R250

Another useful generator uses a shift-register sequence. The implementation known as R250 has several advantages over a linear congruential generator. First, it has a *very* long period, 2^{249}. What is more, this period does *not* depend upon the number of bits used in the random number generator. This makes 16-bit random numbers generated by R250 adequate for many applications. The very long period makes it suitable for scientific applications, such as Monte Carlo and stochastic integration, where many numbers need to be generated. The implementation of this generator is given in Listing Three. The word `r250` generates 16-bit numbers, while `r250d` generates positive 32-bit (i.e., 31-bit) numbers. R250 is also generally much faster to run than an LCM implementation (my 66MHZ '486 generated 30000 `r250d` numbers in 0.55 seconds, and 30000 `lcm_rand` numbers in 0.71 seconds); this also pays off when many numbers need to be generated.

R250 has an overhead of calling another generator 500 times for set-up, so if the set-up time is counted there won't be a speed advantage to R250 when only a small number of values is to be generated.

This generator is built from a *one-bit* random generator that is based upon the equation,

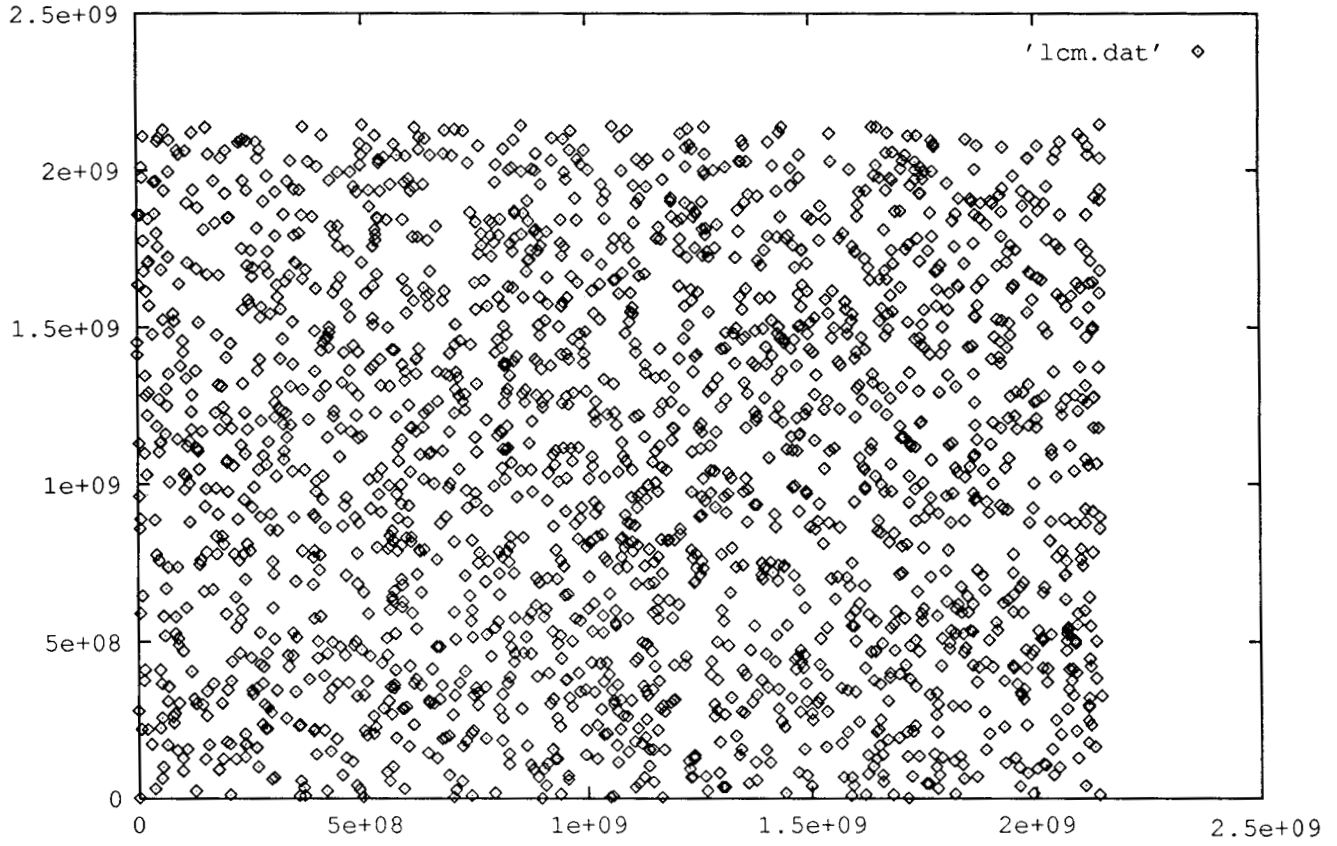
$$I_k = c_1 I_{k-1} + c_2 I_{k-2} + \ldots + c_{p-1} I_{k-p+1} + I_{k-p} mod\ 2 \qquad (4)$$

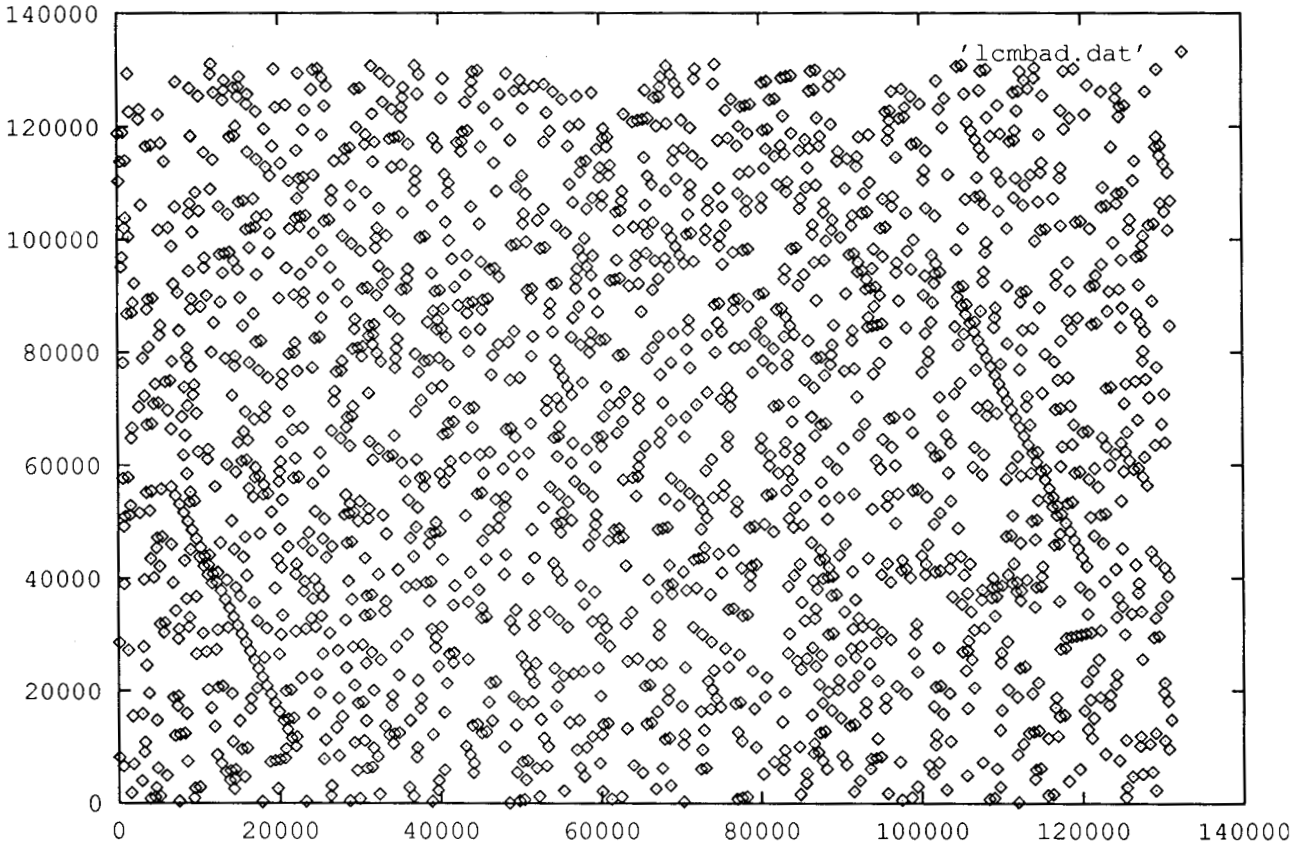
which applies for *each bit.* The maximum period of this sequence is 2^{p-1}, so a large value of p is called for: we will use $p = 250$. We now judiciously choose most of the c_i terms to be zero, so that there are only two terms on the right-hand side,

$$I_k = I_{k-q} + I_{k-p} \qquad (5)$$

and choose $q = 103$. This means then, to generate a random bit, we add the previously calculated 103rd and 250th bits. Now of course, we want to generate a random number of 16 or 32 bits. Obviously this can be accom-

Figure One. Two-dimensional scatter plots of a good (a, upper plot) and a poor (b, lower plot) pseudo-random number generator. 2000 pairs were used in each case.





plished by doing the above one-bit addition for each bit in the desired random number. Noticing that exclusive-or is the same thing as bitwise addition, we can do all the bitwise additions in parallel by using the above equation (5) where the I_k are now *words* and the + is *exclusive-or*. It is this use of exclusive-or as opposed to the multiply and modulus that gives R250 a speed advantage over a linear congruential method.

5. Non-Uniform Distributions

All the generators we have discussed so far attempt to generate uniformly distributed integers over some range. Getting uniformly distributed floating-point values is obviously done by just dividing by the appropriate value. To get uniform integers over a *smaller* range than the full range of a canonical uniform integer generator cannot be done by just taking the modulus with respect to the desired range—that will change the distribution of the results. This is especially noticeable if the desired range is nearly the size of the base range. If the desired range is not too big, one can use the shuffling technique that is described below. For large ranges, unfortunately one has to resort to using floating point and multiplying by the ratio of the ranges and then truncating the fraction. What about obtaining distributions other than uniform? There are many ways of solving this problem (see, for example, Rubinstein, 1981, for an extensive discussion of this topic) but we will only go into two important methods here.

If we have an equation that describes our desired distribution function, then it is possible to use some rather obscure mathematical trickery involving something called inverse probability to obtain a transformation function. This transformation function takes random variables from one distribution as inputs and outputs random variables in a new distribution function. Probably the most important of these transformation functions is known as the Box-Muller (1958) transformation. It allows us to transform uniformly distributed random variables to a new set of random variables with a Gaussian (or Normal) distribution,

$$y_1 = \sqrt{-2 \ln x_1} \cos 2\pi x_2$$

$$y_2 = \sqrt{-2 \ln x_1} \sin 2\pi x_2 \qquad (6)$$

We start with *two* random numbers, x_1 and x_2, which come from a uniform distribution (in the range from 0 to 1). Then apply the above transformations to get two new random numbers which have a Gaussian distribution with zero mean and a standard deviation of 1.

The equivalent *polar* form of this transformation is:

```
: POLAR_BOX_MULLER
   begin
      \ get two random numbers in
      \ the range -1 to 1
      2 ifloat ranf f* f1.0 f-
      2 ifloat ranf f* f1.0 f-

      \ calculate the sum of the squares
      fover fdup f*
      fover fdup f*
      f+
      fdup
      f1.0 f< not       \ is sum >= 1 ?
   while
      fdrop fdrop fdrop \ if so try again
   repeat
   fdup fln fswap f/ -2 ifloat f* fsqrt

   \ multiply the two random numbers by
   \ above factor
   fswap fover f*
   f-rot f*
;
```

The polar form also takes two uniform floating-point numbers in the range 0 to 1, from the routine `ranf` as inputs and returns two Gaussian floating-point numbers (on the float stack) as output. (The code for `ranf` is in `RANTST.SEQ`). The polar form is faster because it does the equivalent of the sine and cosine geometrically without a call to the trigonometric function library. It also is somewhat more robust to numerical problems that can occur when the result of `ranf` is very close to 0.0 or 1.0 (this *will* eventually happen to you if you are generating millions of numbers).

The Gaussian distribution is of major importance in modeling natural systems. This is chiefly because of something called the central-limit theorem. This theorem holds that if a given process is the result of the sum of many other processes, then that process will tend towards a Gaussian distribution as the number of processes composing it gets large. There are some restrictions for this theorem to hold, but these restrictions are usually (but *not* always!) satisfied.

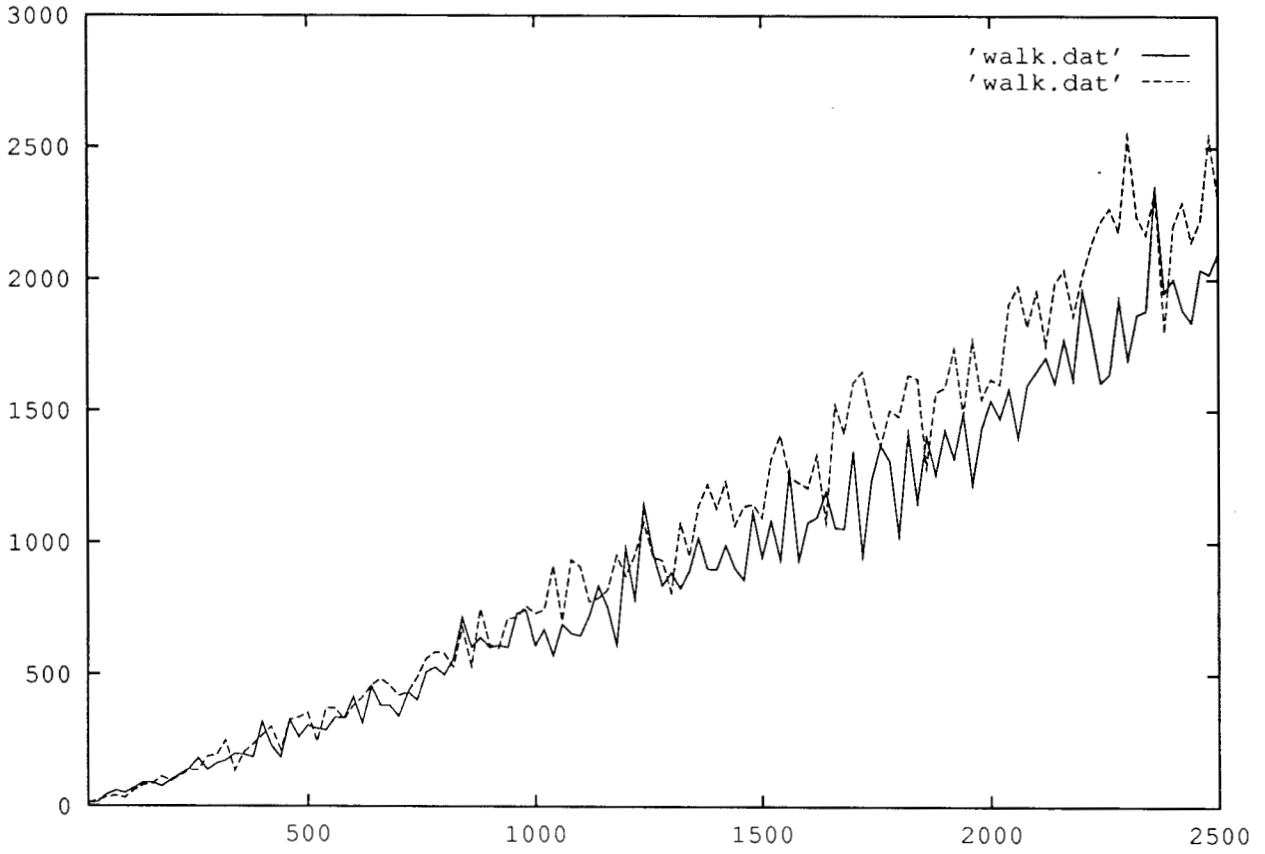
There are other very important distributions as well; in particular the Erlang, exponential, and hyper-exponential distributions are suitable for describing queuing problems such as those that occur in modeling computer and communication systems. Transformations for these distributions, based upon inverse probability, can be found in the literature (see for example, MacDougall, 1987).

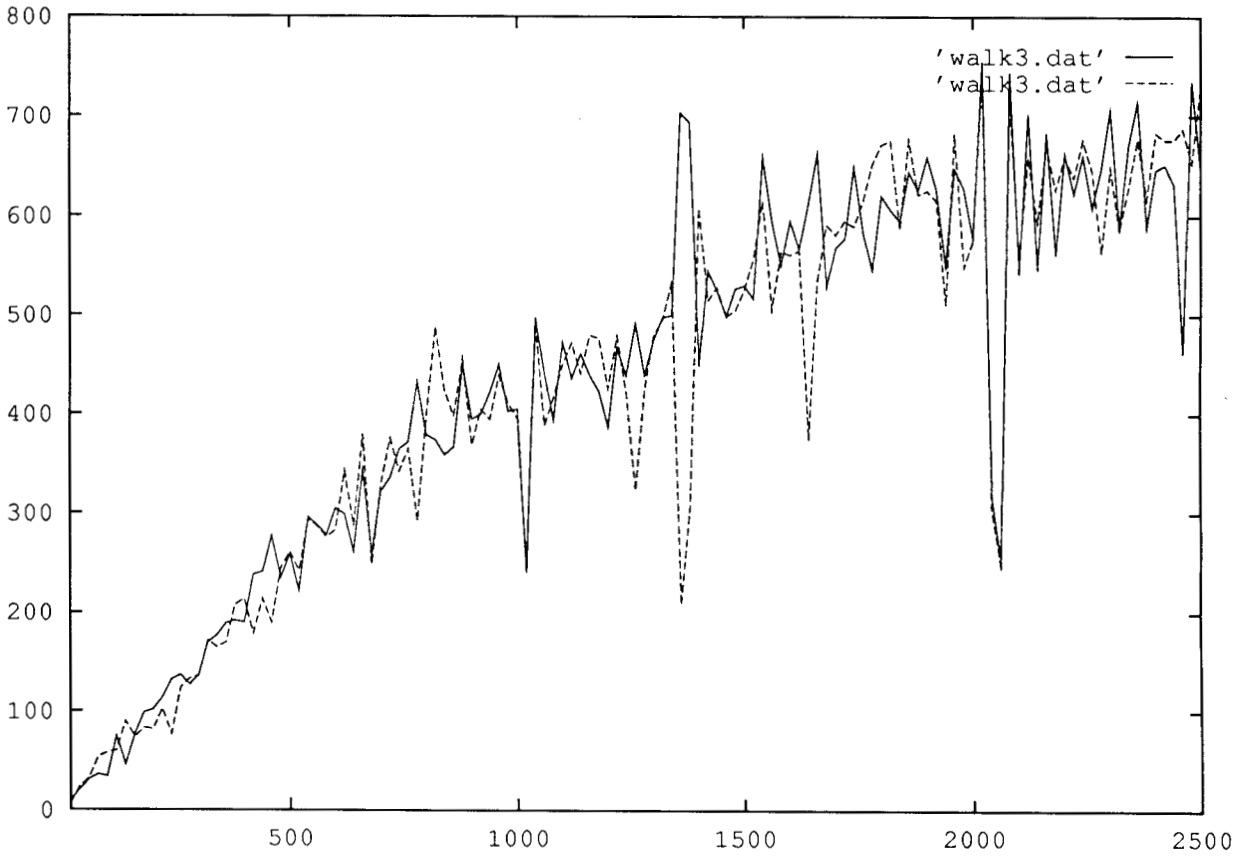
Finding transformations like the Box-Muller is a tedious process, and in the case of empirical distributions it is not possible. Another technique for obtaining values from a different distribution is due to John von Neumann; it is known as the *rejection method*.

The rejection method is basically a "dart throwing" approach to solving the problem. In one dimension, to generate a random number from a distribution $f(x)$, one generates a trial random number, x, from a known distribution, $p(x)$. The distribution $p(x)$ must be such that for any x, the probability $p(x) \geq f(x)$. Usually $p(x)$ is taken to be uniform, but it is not necessary. A second random number is generated from a distribution that is uniform on the possible range of the desired distribution; call this number y. If the value y is less than or equal to $f(x)$, then we accept x. If y is greater than $f(x)$ then we reject it.

The rejection method is very powerful, since it can be applied to distributions that are known only in tabular form, as well as to analytic distributions. It is also easy to implement in any number of dimensions. If the domain of the desired distribution is infinite (like the Gaussian distri-

Figure Two. The mean square distance from the starting point for the X (solid) and Y (dashed) components for a two-dimensional random walk. Each curve is the average of 50 independent walks. (a, above) is an example of the result for a good generator (R250); the plot is roughly linear. (b, below) is an example from a poor generator (LCM_BAD); note how the plot changes slope after about 900 steps.





bution), then a finite truncation approximation must be used. Depending upon the application, this may or may not be a problem. In the example of the rejection method shown in Listing Two (`gauss_test`), we generate Gaussian random numbers (with zero mean and unit standard deviation) using the rejection method. We have assumed that the application is not sensitive to values above 4.0 and below -4.0. Values like these *should* occur on the average once in ten thousand times, but the rejection method will *never* produce them. The biggest problem with the method is that it can be inefficient, in the sense that many random numbers need to be generated in order to keep only a few of them. The efficiency is controlled by the ratio of the volume of the starting distribution to the volume of the desired distribution. If a uniform distribution is used to start with, then the starting volume is the volume of the box that completely bounds the desired distribution. If the desired distribution has tall peaks, the resulting ratio can be very small (I have used applications where the ratio was as small as 10^{-5}). In two dimensions, the *best* possible efficiency is 50% (one good number is produced for every two numbers generated). This would imply that an acceptable random number was generated each time. This gets worse in higher-dimension problems. The efficiency for the one-dimensional truncated Gaussian distribution in the above example is 15.7%.

In most of the methods used to get a given distribution function, the easiest starting distribution is uniformly distributed random variables. Therefore, the ability to reliably generate uniformly distributed random numbers is fundamental to applications that use random numbers, no matter what distribution function is appropriate to the problem at hand. This is the reason that so much effort is put into the building of a canonical uniform integer generator. *[Continues in next issue.]*

Everett Carter is an Assistant Professor of Oceanography at the Naval Postgraduate School. Prof. Carter wrote the Forth system for and helped design the RAFOS float which is being used internationally as part of the World Ocean Circulation Experiment. Back on land, he generates several billion random numbers a week running stochastic models of oceanic currents.

Listing One. STATS.SEQ

```
\ stats.seq      Calculates statistics (chi_2 and correlation)
\                of files of numerical data
\                NOTE: Uses floating point numbers in their own stack
\     (c) Copyright 1994  Everett F. Carter. Permission is granted by the
\     author to use this software for any application provided the copyright
\     notice is preserved.

: stats_task ;

needs ffloat.seq
needs fileio.seq
needs rands.seq              \ need to have maxrand

cr .( STATS.SEQ    V1.1                    1/2/94   EFC )

decimal

: fiarray  ( n -- )           \ like farray, except array is initialzed to 0
        create
           dup , 0 do f0.0 f, loop
        does>
          swap dup 0<
                if drop @
                else
                   8 * 2+ +
                then
;

\ code for calculating the autocorrelation

50 constant maxlag

maxlag 1+ fiarray data_buffer
variable lag_value

fvariable sumx          fvariable sumy
fvariable sumxy         fvariable sumxx         fvariable sumyy

: buffr_shift ( -- )
        lag_value @ 0= if exit then

        lag_value @ 1+ 1 do
                  i    data_buffer f@
                  i 1- data_buffer f!
```

```
        loop
;

: xvalue ( -- addr )
        lag_value @ data_buffer
;

: yvalue ( -- addr )
        0 data_buffer
;

: cor_init ( n -- n )

        dup lag_value !

        f0.0 sumx f!   f0.0 sumy f!
        f0.0 sumxx f!  f0.0 sumxy f!     f0.0 sumyy f!
;

: calc_cor ( n --, f: -- c )     \ calculate the correlation at given lag

        cor_init

        dup 0= not if
                        \ read in the first lag points
                        1+ 1 do read_float 0= if abort" read error" then
                        i data_buffer f! loop
                else
                    drop
                then

        0              \ start data count

        \ read in and accumulate the correlations
        begin
          read_float
        while
          buffr_shift
          xvalue f!
          xvalue f@ sumx f@ f+ sumx f!
          yvalue f@ sumy f@ f+ sumy f!
          xvalue f@ fdup f* sumxx f@ f+ sumxx f!
          yvalue f@ fdup f* sumyy f@ f+ sumyy f!
          xvalue f@ yvalue f@ f* sumxy f@ f+ sumxy f!

          1+            \ increment data count

        repeat

        \ now form the actual correlation
        dup

        sumxy f@  sumx f@ sumy f@ f*  ifloat f/ f-

        dup
        sumxx f@ sumx f@ fdup f* ifloat f/ f-

        sumyy f@ sumy f@ fdup f* ifloat f/ f-

        f* fsqrt

        f/
;


: cor ( n -- )                  \ calculate and print correlations up to lag n

        dup maxlag > if ." cor: correlation lag must be <= "
                        maxlag . cr abort then
```

```
        seqhandle+ !hcb
        seqhandle+ hopen abort" file open error"

        cr

        1+
        0 do i .
             i calc_cor f. cr
             hrewind
       loop

        seqhandle+ hclose abort" file close error"
;


\ code for calculating the chi-square

50 constant maxbins

maxbins fiarray observed
maxbins fiarray expected
fvariable  factor

: set_factor  ( bins -- )
        ifloat maxrand float f/ factor f!
;

: calc_chi_2  ( n --, f: -- c )        \ calculate the chi-squared

        f0.0
        0 do i expected f@  i observed f@ fover
             f- fdup f*
             fswap f/
             f+
        loop
;

\ accumulate histogram from data file
: hist_accumulate ( bins -- count )      \ count is the number of data points

        dup 0 do f0.0 i observed f! loop

        set_factor

        \ count them up, leaving number of data points on the stack
        0
        begin

                \ get the float number from the file on the (float) stack
                read_float
        while
                \ calculate the class interval -- 0 assume to be minimum
                factor f@ f* int drop
                observed dup f@ f1.0 f+
                f!

                1+
        repeat
;

: set_uniform ( bins count -- )        \ set expected to count/bins

        ifloat
        dup ifloat f/                  \ calculate the expected number

        0 do
               fdup i expected f!
        loop
```

```
        fdrop
;

: verify_observed ( bins -- )

        dup
        0 do i observed f@ f0.0 f=
             if cr . ." bins are too many, bin " i . ." is empty "
                cr abort then
        loop

        drop
;

: chi_2  ( bins --, f: -- c )

        dup maxbins > if ." chi_2: number of bins must be <= "
                         maxbins . cr abort then

        seqhandle+ !hcb          seqhandle+ hopen abort" file open error"

        dup dup
        hist_accumulate

        \ check that observed has no zeros at this point
        over verify_observed

        set_uniform
        calc_chi_2

        seqhandle+ hclose abort" file close error"
;
```

Listing Two. RANTST.SEQ

```
\ rantst.seq      code to set up tests of random number code
\     (c) Copyright 1993  Everett F. Carter. Permission is granted by the
\     author to use this software for any application provided the copyright
\     notice is preserved.

: rantst_task ;

needs ffloat.seq
needs fileio.seq
needs rands.seq

cr .( RANTST.SEQ   V1.1                12/28/93   EFC )

decimal

: lcm_test_loop   ( -- )
             1. seed 2!          \ set initial seed value
             1.                  \ push a temporary value
             10000 0 do 2drop lcm_rand loop
                      ." final value: " d.
                      ." should be 1043618065" cr
;

: shuffle_test   ( n -- )        \ shuffle n elements and display result
        1. rand_init

        dup ramp                 \ set the initial sequence
        cr
        dup 0 do i 20 mod 0= if cr then   \ show the initial sequence
             i s@ .
        loop
        cr

        dup shuffle                        \ shuffle the sequence
```

```
        0 do i 20 mod 0= if cr then          \ show the result
             i s@ .
        loop
        cr
;


: test_rand   ( n -- )        \ write out n pairs of random numbers
      1. rand_init

      seqhandle+ !hcb
      seqhandle+ hcreate abort" file creation error"

      ['] htype is type
      ['] hcrlf is cr

         0 do randgen  u.    3 spaces
              randgen  u.    cr
           loop

        ['] (type) is type
        ['] crlf is cr

        seqhandle+ hclose abort" file close error"
;

: test_quasi ( n -- )
      2 quasi_init

      seqhandle+ !hcb
      seqhandle+ hcreate abort" file creation error"

     ['] htype is type
     ['] hcrlf is cr

         0 do quasi  0 ix 2@   ud.    3 spaces
                     1 ix 2@   ud.      cr
           loop

      ['] (type) is type
      ['] crlf is cr

        seqhandle+ hclose abort" file close error"
;

: test_drand   ( n -- )        \ write out n pairs of 32 bit random numbers
      1. rand_dinit

      seqhandle+ !hcb
      seqhandle+ hcreate abort" file creation error"

      ['] htype is type
      ['] hcrlf is cr

         0 do drandgen  ud.    3 spaces
              drandgen  ud.      cr
              loop

        ['] (type) is type
        ['] crlf is cr

        seqhandle+ hclose abort" file close error"
;


\ code for generating a random walk
16384   constant qrtr
32767   constant half
49151   constant threeqrtr
```

```
variable xpos           variable ypos
2variable xsq           2variable ysq

: xinc   1 xpos +! ;            : xdec  -1 xpos +! ;
: yinc   1 ypos +! ;            : ydec  -1 ypos +! ;

: rwalk  ( -- )            \ do a single random walk step

        randgen

        dup qrtr      u< if xinc else
        dup half      u< if yinc else
        dup threeqrtr u< if xdec else
                         ydec then then then
        drop
;

: walk ( n m -- E<y^2> E<x^2> )          \ n -- number of steps per walker
                                 \ m -- number of walkers to accumulate over
                                 \ returns average xsq and ysq

        0. xsq  2!        0. ysq  2!

        swap over
        0 do
            0 xpos !        0 ypos !
            dup 0 do rwalk loop
            xpos @ s>d 2dup d* xsq d+!
            ypos @ s>d 2dup d* ysq d+!
       loop
       drop

       dup
       xsq 2@ rot m/mod >r drop
       ysq 2@ rot m/mod swap drop
       r>
;

: walk_test ( n m -- )           ( n -- maximum number of steps per walker )
                                 ( m -- number of walkers to average )
        1234. rand_init

        seqhandle+ !hcb
        seqhandle+ hcreate abort" file creation error"

        cr
        swap
        20 do
           i .

           i over walk

           ['] htype is type
           ['] hcrlf is cr

           i u.      3 spaces
             u.      3 spaces
             u.        cr

           ['] (type) is type
           ['] crlf is cr

           cr
        20 +loop
        drop

        seqhandle+ hclose abort" file close error"
;
```

```
\ Test of Monte Carlo integration
\ integrate 2-d function func() from 0 - 1, 0 - 1

: func ( --, f: x1 x2 -- y )        \ test function x1^2 + x2^2

        fdup f* fswap fdup f* f+
;

: fquasi ( n -- , f: -- x1 x2 ...xn )    \ form floating point values
                                         \ in range from 0 to 1
       quasi

       0 do
            i ix 2@ float quasi_max float f/
         loop
;

: mcint_test ( iters --, f: -- x )
     2 quasi_init

     f0.0

     dup 0 do  2 fquasi  func  f+ loop

     ifloat f/
;

\ generate (an approximation to) Gaussian random numbers using the
\ rejection method.  Will not produce values below -4.0 or above 4.0
\ which SHOULD occur on the average once in 10,000 times

FLOATS

2.50663 fconstant fnorm
-4.0    fconstant lbound
 4.0    fconstant ubound

: ranf ( --, f: -- x)             \ generate a random value from 0.0 to 1.0

        drandgen float maxrand float f/
;

: gauss ( -- , f: x -- y )      \ calculate Gaussian pdf at given value
        fdup f* f2/
        fnegate
        fexp

        fnorm f/
;

: trial_value ( --, f: -- x )            \ form trial value
        ranf
        ubound lbound f-
        f*
        lbound f+
;

: gauss_rand ( --, f: -- x )          \ generate a Gaussian random variate
                                      \ using the rejection method
        begin
          trial_value
          fdup
          gauss

          ranf f<
        while
          fdrop
        repeat
;
```

```
: gauss_test ( n -- )           \ write n Gaussian numbers to file

        seqhandle+ !hcb
        seqhandle+ hcreate abort" file open error"

        ['] htype is type
        ['] hcrlf is cr

        0 do
           gauss_rand  f.  cr
        loop

        ['] (type) is type
        ['] crlf is cr

        seqhandle+ hclose abort" file close error"
;


\ Differential equation example
\ solve for the steady state temperature distribution within an annulus

fvariable xi
fvariable yi
0.1 fconstant scale


: laplace  ( maxit -- , f: x y -- t )

        1. rand_init

        yi f!   xi f!

        f0.0

        \ loop over maxit walks
        dup 0 do
                xi f@ scale f/ fix drop xpos !
                yi f@ scale f/ fix drop ypos !

                begin
                        rwalk

                        \ determine current radial position
                        xpos @ dup *
                        ypos @ dup * +
                        ifloat fsqrt

                        \ account for scaling
                        scale f*

                        fdup fdup 3.0 f> 1.0 f< or not
                 while
                        fdrop
                 repeat

                 \ at this point we are either at r < 1.0 or r > 3.0
                 3.0 f> if   60.0
                        else 40.0  then

                 f+

        loop

        ifloat f/
;
```

Pygtools—

A Library of Reusable Utilities

L. Greg Lisle
Winston-Salem, North Carolina

Reading the available literature on Forth, and especially the comments on the Forth-Net, one often hears of the need for reusable libraries of tools. Unfortunately, there seems to be a companion lack of these libraries. The most common call is for the ability to access C libraries from Forth. This seems a rather unsatisfactory solution.

As an alternative, I would like to propose a library structure designed for use with Forth, where the programmer can do whatever he or she wishes. In addition, this package demonstrates the flexibility that comes with using a screen-based disk structure. I wrote this package using Pygmy Forth largely because of the clean disk interface, but the concepts should apply to any Forth that provides block access.

The Pygtools

What follows is a brief description of the contents of the Pygtools library, and an exposition on the utilities I have created to make the library (or any other similar library) more useful and accessible.

As Screens 3–9 of the sample listing indicate, Pygtools covers a broad range of tools and utilities. Using the DOCUMENT utility, you will find the library has close to a thousand definitions. They cover a wide range of topics including: basic tools, F83 compatibility, double number math, advanced integer math, system utilities, advance terminal I/O, debugging tools, and hardware device drivers. There are a couple of CASE structures, DO ... LOOP, secondary stacks, and string handlers.

Integer math functions include: trig, Log2, x^n, random numbers, interpolations, Julian dates, factoring, and BCD conversion. System utilities include: read-only file opening, function key translations, DOS environment access, file attribute handling, filename input and construction, and hard copy utilities. I/O includes BOX definitions for the screen, calendar functions, time I/O, Soundex, and number output in English.

Debugging tools include an advanced decompiler, a single-step utility, a breakpoint function, divide-by-zero protection, extended DOS error display, and a display of all the words that use a given word. There are hardware drivers for the display, joysticks, sound (w/PC speaker), music, mouse, keyboard, COM ports, and timers in both hardware and software.

Also included are some experimental words for supporting overlays. If these aren't enough, two additional packages are available.

New Library Prototype

Pygtools may be viewed as a prototype for a new style of Forth library. I did not have a concrete set of design goals when I started the project, and I doubt that what I

```
( SAMPLE.SCR  V1.3  Library utilities & tools        12Nov93LGL)
( This block file contains a sampling of the tools created by
  L. Greg Lisle and is copyrighted by him in 1993.  I hereby
licence anyone to use them and distribute this sample, so long
as this attribution screen is included.
  The Basic PYGTOOLS package is shareware and is available in
the GEnie Forth Board Library.  The Full PYGTOOLS library and an
Advanced Tool Library are available for purchase from me at the
address below.
  If you have any comments, complaints, bug reports, requests or
whatevers, you can contact me at the following address or by
     E-Mail on Internet as      L.SQUARED@GEnie.geis.com
     Greg Lisle   2160 Foxhunter Ct    Winston-Salem, NC 27106 )

 ( STANDARD LOADER                                      12Nov93LGL)

     BLK @  >UNIT#   DUP  1000 *  DUP  SCR !     $C000 SET-EDGE

          15 +   LOAD     ( ?LOAD & ULOAD )     REPORT OFF
             14 ULOAD     ( DIR, FROM & GET )

          GET TUCK
```

```
( Environmental Comments: )  ." No Target "    ( 12Nov93LGL)

( You can edit screen 1 to include the tools you most often use.
   I personally use the following standard SCR usages:
0 description, 1 development LOAD, 2 target LOAD,  3-5 index
or 3-5 could be optional utilities or LOAD screens.
   Therefore, I automatically load screen u001 when I open a
source file.  This loads any utilities I use for development.
I then load screen u002 to create a target.COM file for
production.
   In general, my tool libraries will have indices on screens
3 through whatever, while application code will have additional
load screens. )

( Loadable Index   { use with GET }   Drivers      12Nov93LGL)
10  >A-BLK           11  FROM            13  GET
14  DIR              15  ?LOAD           16  EXIT?  Optionals
 Device Drivers
18  VEMIT            19  BLINK           20  CYAN   colors
21  VMODE            22  dWIDE   132 col  23  dL   double List
24  PIX!   pixel !   25  PIXROW          26  PIXBOX
28  BJOY       BIOS  29  JOYSTICK  noBIOS
30  TONE  Sounds     32  PLAY Music      35  TUNE:
36  ESC# for printer 37  PITCH printer   38  PRN-STAT
39  ?MOUSE           41  MOUSEXY         42  PWM
45  BKEY  F11, F12   46  #KEY   only     47  CAPS-ON  off etc
48  CPORT Serial     49  DXR  wait for in 50  COMM
56  DOS-EMIT
57  ETIME   elapsed  58  TIMER  in Secs  59  HWTIMER  HW time
17  RECORD@ disk I/O

( Loadable Index   { use with GET }  Basic Tools   13Nov93LGL)
16  TUCK             17  UNDO
 Basic Tools
60  TUCK  etc        61  CUR+  etc       62  1+!   etc
63  PICK  & ROLL     64  COMBINE & SPLIT 65  -ROT   & -ROLL
66  ARGUMENTS        68  DIGIT?          71  LATEST  DO  LOOP
72  2SWAP  2OVER     73  2PUSH  2POP     74  2SPLIT
75  2CONSTANT        76  D-AND   D2^     77  BROLL  Bit shft

78  ARRAY            79  VARRAY
80  T:  Table Build  82  ASTACK  Aux     83  TSP  Stack

84  bcase            85  godo            86  ECASE
84  BCASE            85  GODO

87  SCAN$            88  LOOKUP          89  SCAN$2

( Loadable Index   { use with GET }  I/O            5Nov93LGL)
 Advanced Terminal I/O
90  BOX$             93  BOX:  creator   94  BCLS  scroll
95  BINIT  boxes     97  BSIZE  BPUT  BGET
98  UC>lc            99  NUM$            100 VAL
101 T.    temp conv  104 $reENTER        105 #reENTER
106 $INPUT           107 SOUNDEX

108 DAYS  Calendar   109 .CALENDAR
110 DOW/MON adv cal  111 .DATE  date out
112 UNTIME  DOS dcod 113 TIMEIN & Out

114 DNUMBER D Input  115 D.  D Output    116 ,D.  Comma .
117 .CARDINAL        118 .ENGLISH numbers
119 B.  bit print
```

have achieved thus far is the final form it will take. Given those caveats, these are the features I am currently highlighting:

1. Ease of access to tools
2. Tool file can be modified without changing access
3. Access to tools is more comprehensible than standard LOAD
4. Tools are provided as source code

Item one is provided by the words FROM, PREVIEW, GET, and their variants. Their syntax is as follows:

```
FROM toolfile.scr
PREVIEW toolgroup
GET toolgroup
```

FROM designates the tool file to be used until further notice. If the file is open, the whole filename is not needed, FROM will match a partial string as well. It will also check both upper and lower cases. If the library file is not currently open, FROM will try to open it. E.g.,

```
PYGT ==> PYGTOOLS.SCR
pygm ==> PYGMY.SCR
```

The operation of PREVIEW and GET are similar, and use the same search code. Given a tool-group name, they use a file index to locate the screen on which that group is defined. PREVIEW then lists the file, while GET loads it. Obviously, for this to work, the index must be in a known location and of a known format. The convention I am currently using is to start the index on the second line of the fourth block in the file, and to continue with as many blocks as needed. Line zero of each block is reserved for comments, labels, and date stamp. The format is three entries per line, consisting of a local block number, the group name, and a short label.

If, as is done in the Pygtools, the group label is one of the words in the group, another access tool is also possible. One problem that arises with complex applications using a large collection of reused code, is reloading something that is already resident. To prevent this, I created the word ?GET. ?GET looks like a normal GET, but in action it first looks to see if the tool-group name is currently in the dictionary. If it is not found, ?GET proceeds like a normal GET; otherwise, it skips to the next operation. This allows loading interrelated tools in any order, with no reloading of subtools.

A recent addition is a switchable EXIT. If the variable ?EXIT is off, the exit is skipped, thereby loading the whole screen. To use this, I also added LOADALL, GETALL, and ?GETALL to disable the EXIT while loading a screen.

Item two is accomplished by using a separate index to the tool groups. Thus, blocks can be added, moved, even shuffled, but GET will still load the correct block. This means that programs that use GET to access the library will not need to be updated. The index could be manually updated, but I have created a REINDEX function to update the pointers automatically. (I will often rearrange the index manually after a reindex, but this is done for clarity and aesthetics.) To support the REINDEX function, I use an additional convention. All blocks have comments in their first line, but only tool-group load screens have the opening parenthesis in the first byte. Further, the first word in the comment is the tool-group name. As just implied, a tool group is loaded by loading a single block. If the group uses

```
( Loadable Index   { use with GET }  Utils          12Nov93LGL)
 System Utilities
120 >FILE  & Rd Only 121 BCOMP block Comp 122 AUNIT
123 INDEX  of blocks 124 CHANGES          126 Q uick Index
127 BLK>PG           128 VOC?
129 $TAG scr w/ date 130 NEWBOOT          131 FENCE  & EMPTY
132 ADD$  string bld 133 ADD-DATE  more   134 WORDS>  file
135 +EXT   bld file  136 USE  & WO Work On
137 GETFN            138 UNDO & VIEW      139 COPY+
140 BUILD            141 LDUMP  LC@+      143 ENVIRON (DOS)
144 F05 functionKeys 147 F01   more Fkeys
148 FTYPE            149 F-ATTRIB         161 ?FMAKE
150 GETCHAR          151 RESEED  encrypt   152 $WORD
153 $+$  concat.     154 $.R   Field .     155 MID$   extract
158 S&R   Srch&Repl  159 CODE-SRCH         160 DOCUMENT

( Loadable Index   { use with GET }   Math           5Nov93LGL)
 Math  Tools
162 D+  D-  S->D     163 D=  D<  DMAX
164 UMD*             165 MD*  UD*  D*      166 D/MOD  D/  DMOD
167 UM*  2/MOD MU*   168 M/MOD   /MOD

169 SQRT             170 BCD>Bin  & back   171 RANDOM
172 Log2  x^n        173 AFACTOR  primes

174 JD  Julian dates 175 J>YMD  inverse    176 INTERP OLATE

178 SIN    COS       179 ATAN2

( Loadable Index   { use with GET }   Debug &       12Nov93LGL)
 Debug Tools
181 .XID             185 SEE:  decompiler
186 USAGE            187 USERS            188 ORPHANS
189 .S   special .S  190 .SV  vertical    191 SS@
192 GUARD memory     193 ?/0  Divide prot 194 .T   tracer
195 XABORT more info 196 SOFT-ABORT       197 MY.ID print
198 STEP  one word   207 BREAK-ON  points
210 DOS4   ErrCode   211 DFREE  status    212 SPEED   of CPU
213 .STATS           214 .EQUIP
215 .COMSTAT  RS232  216 .ERROR  code     218 DSTAT
219 HEXED  mem edit
222 EHELP2  simple   223 HELP   from Disk 225 DUMMY words
227 REBOOT  Cold     228 >ASCII display   229 >PAD"  $ to PAD

( Loadable Index   { use with GET }   Extras        10Nov93LGL)
 Advanced Development
231 ALLOC            232 SET-BUFF         236 OVERLAY:
                       Full Pygtools
246 ?DO              249 C+!    & misc    250 DCONVERT
252 FGET$            253 FINDIT           254 BSWAP
255 ≤"     alt243    257 ATABLE           258 10*
261 >FILE"
267 DTA   Tools      268 GETDRV           269 GETDIR
270 DOSDIR           271 VOL-LABLE
273 READ-SECT
279 SSORT  sort fcn

283 REINDEX          284 UNINDEXED

0   *END*

( >A-BLK      Active Unit Handling                  12Nov93LGL)
: B>BASE ( n-n')  >UNIT#   1000 *  ;
```

```
  VARIABLE AU          BLK @   B>BASE   AU !   ( Active Unit )

: >A-BLK ( n-blk#)  AU @  B>BASE  2DUP  0=  SWAP  3 =   AND
                                          189 AND   + + ;
| : o>a   ( o - a)  AU @  BLOCK  +  ;

: Cswap ( a-) COUNT FOR  DUP C@  $20 XOR OVER C! 1+ NEXT  DROP ;

: SAME? ( a u - f)  FNAME @ 1+  SWAP  2DUP  COUNT  COMP
       IF  DUP  Cswap  COUNT COMP  ELSE   2DROP 0   THEN  0= ;

( FROM   Select Tool File     UNIT#<                  12Nov93LGL)
                                    -1 +LOAD

: UNIT#<  ( a-n)      -1
          BEGIN  MAX-FILES 1+  OVER  >  WHILE
                    1+  2DUP  SAME?  UNTIL   THEN     NIP ;

: FROM  HERE 20 0 FILL  32 WORD   DUP  UNIT#<   DUP MAX-FILES >
          IF   DUP  ?CLOSE   2DUP OPEN   THEN
          1000 *  AU !   DROP ;

EXIT
Example:  FROM  PYGT     GET  TABLE    GET  INTERP    GET  SQRT
          FROM  BRADTOOL.SCR     PREVIEW  XDUMP


   ( GET tools                                        12Nov93LGL)
                      11 ?ULOAD  FROM

| : GNEXT ( o-o' a) 21 +  DUP  1022 >
                          IF  1 AU +!   960 -  THEN
                          DUP  63 AND  10 <  -    DUP o>a  ;

| : ThisIt? ( a a-f)  COUNT  COMP 0= ;
| : Finish ( a - f)   " *END* "  ThisIt? ;

| : B# ( a-n)  BASE @   PUSH   DECIMAL
                    3 -TRAILING  (SNUMBER >A-BLK   POP BASE ! ;

: SEARCHDEX ( a-o f)    PUSH   3 >A-BLK  AU !   47   0
               BEGIN  DROP  GNEXT  DUP R@  ThisIt?
                   SWAP  Finish  OVER  OR UNTIL    POP DROP ;


( GET   PREVIEW       use Index                      12Nov93LGL)

               -1 +LOAD  (   Tools )

: GET# ( a-a b#)  DUP  C@ 0= ABORT" No ID "   DUP  SEARCHDEX
             IF  4 -  o>a  B#
              ELSE   DROP  TYPE$  1 ABORT"  Not Found "  THEN ;

: GET  32 WORD   GET#   SWAP  'g  ALoad    LOAD ;

: PREVIEW  32 WORD  ( OVER TYPE$  SPACE)  GET#  SCR !  DROP  L ;

: ?GET  >IN @  CONTEXT @   -'  NIP
                IF  >IN !  GET  ELSE  DROP  THEN ;


( DIR     Print a 2 Column Directory                  12Nov93LGL)
                                      13 ?ULOAD  GET
| : D-ONE ( a-)   DUP  1- C@   32 >
```

more than one block, or requires other, subsidiary tools, they are loaded by that load block. To maintain location independence, I use either ?GET or +LOAD to load additional blocks.

Item three is inherent in the GET function. Using the form GET 2SWAP instead of 2072 LOAD makes a reading of the load screens easier to understand. What is being accomplished is obvious from the code itself.

Item four is both an advantage and a disadvantage. It is a disadvantage to the toolsmith, in that it reduces the control over the tool library. It is an advantage to the programmer by allowing customization when needed. Given Forth's history, I feel that the latter is more in keeping with "standard practice." Whether it is a fatal flaw remains to be seen.

The adjoining listing includes most of the tools I have described, plus a few extras. REINDEX is included with the full version of Pygtools. The sample file and basic Pygtools files are available in the Forth library on GEnie.

Future development will include a smarter FROM to open the file, an improved DIR to list the index, and perhaps a SUBGET to load just part of a tool group. In addition, I expect to continue expanding the contents of the library and will consider submissions or requests from the Forth community. As an alternative, if you wish to use the library structure described here to create a specialized library, please do so. Subjects could include advanced graphics, floating point, complex math, matrices, or whatever.

Greg Lisle is an E.E. with over ten years of working with Forth, and 18 years with microprocessors. He is currently building a consulting practice in North Carolina. He may be reached on the Internet at L.SQUARED@GEnie.GEIS.COM.

```
                IF  CR  ." ==== "  4 -  40 TYPE  CR
                   ELSE  DUP   C@  32 >
                      IF   4 -    B#  BLOCK  1+  39 TYPE  SPACE
                        ELSE   DROP  THEN  THEN ;

: DIR   3 >A-BLK  AU !  47   CLS   ." Directory of "
          AU @  >UNIT#  FNAME @  TYPE$   CR
        BEGIN  GNEXT   Finish   NOT     ?SCROLL ( OVER 1 DU)
            WHILE   D-ONE   REPEAT   2DROP ;

  (   EXIT  ( Opt hardcopy )                          ?GET ESC#
: PDIR  RESETPRN  >PRN     AU @  WHD    DIR  CR
                  >SCR  SETPRN ;


( ?LOAD  ?ULOAD    LOAD if not present                4Nov93LGL)

                  VARIABLE REPORT    REPORT ON

| : ALoad ( n a c-n)  REPORT @  IF  2 SPACES  EMIT  ." Loading "
                                   TYPE$  ."  @ "   DUP .
                                 ELSE   2DROP  THEN  ;

: ?LOAD ( n) CONTEXT @ -'  IF   32  ALoad  LOAD
                               ELSE   2DROP   THEN ;
  177  ?LOAD  ULOAD

: ?ULOAD ( n)   CONTEXT @ -'  IF  'U  ALoad  ULOAD
                                 ELSE  2DROP   THEN ;

: +LOAD ( n-)  BLK @  +   "   "   '+  ALoad   LOAD ;


(  TUCK  & other Missing tools   ><  .L   H.  [#60] 10Sep93LGL)

CODE TUCK  ( a b-b a b) AX POP, BX PUSH, AX PUSH, NXT, END-CODE

CODE >< ( n-n')  BH BL XCHG,  NXT,  END-CODE

: L  CLS L 18 70 AT ;
: .L ( scr#-)  SCR ! L ;

: UL ( n-)  SCR @  >UNIT#  1000 * +  .L ;

: VIEW ( -) ' VFA @  ?DUP IF  .L  THEN ;

: H. ( n-)  BASE @  HEX SWAP U.  BASE ! ;


( UNDO & New VIEW     V RV PV    [scr# 138]          15Sep93LGL)

: UNDO   PREV @ BUFFERS  DUP @  $7FFF AND  0 ROT !  .L ;

  VARIABLE  OLDSCR

: VIEW ( -) SCR @  OLDSCR !   32  WORD  2  -FIND
                              IF  4  -FIND     ABORT" ??"  THEN
                              VFA @  ?DUP IF  .L  THEN ;

: V  VIEW ;           ( shorthand )

: RV OLDSCR @  .L ;   ( Review where you were )

: PV  SCR @  OLDSCR !   PREVIEW ;
```

Fast FORTHward

Rapid Development Demands Quality Interfaces

Mike Elola
San Jose, California

With our productivity hanging in the balance, interface design is a significant concern. Well-crafted interfaces are required between routines, not just between software and its users (the so-called "user interface").

We like to think of Forth as a means of increasing our productivity. Forth is part of the solution, not part of the problem. Our preoccupation with increased productivity can be Forth's competitive advantage—but only if Forth vendors and Forth programmers steadfastly make this their goal.

Even Forth systems as popular as F83 have reduced our productivity at particular times. In this issue of *FD*, you will come across an article by Byron Nilsen warning us about some of the pitfalls of the F83 vocabulary mechanism.

Forth productivity is not a matter of creating Forth systems that are mostly of sound design. Forth systems must be fashioned from *comprehensively* sound designs. Anything less will undercut our claims to rapid development. Considering that the loss of one day's work for 500 Forth programmers is two-man years of Forth programming labor misspent, the need for high-quality development systems is obvious.

Nothing is more humbling than to be the primary user of your own software creations...

The burden for this responsibility falls in the hands of the Forth vendors. Their labors will determine if the reputation of Forth as a rapid development tool withers or grows. Articles like Nilsen's need to be written and published to motivate all of us to design in a user-centric fashion. (In this case, Forth programmers are the users.)

I would also like to see guidelines published that can help us produce consistently behaved and easily learned sets of routines. In keeping with the philosophy that he who asks for something must volunteer to provide it (which is especially true for FIG), I'll try to intelligently discuss certain interface issues. For starters, I'll examine the technique of using variables to help serve as the interface between routines.

Using Stack-Buffered Variables

There is at least one known way to use variables to help parameterize routines yet still enjoy interfacing advantages such as reentrancy. To do this, use a stack as a buffer for older input variable states. This is how Forth makes use of the return stack, which stores the states of instruction pointers for all routines that are underway—with the exception of the currently executing routine.

A similar approach is used to create a new text interpretation stream by pushing the old `>IN` and `BLK` values onto return stack, resetting their states for the new stream, and then letting interpretation of the new stream end before restoring the old states of `>IN` and `BLK`.

(This technique differs from the use of local variables in small ways. The scope of a local variable is more restricted. It cannot be referenced in several routines, as you can a stack-buffered `BLK` variable. In accordance with the way that many local variables are initialized, the values contained in `BLK` start out as stack parameters. However, the stack-orientation of local variables is hidden from view most of the time. Many experts view such encapsulation actions as the way to make programs more readable.)

Sensitivity to the Calling Context

Stack buffering of variable values helps preserve vital information that, if lost, would cause a loss of program synchronization. For example, if I request the interpretation of block 81 and, halfway through, another request in the input stream caused block 91 to be loaded, the calling context information about loading block 81 must be preserved correctly by `LOAD` so that (1) the first half of the block 81 is not interpreted twice, and (2) the second half of block 81 is not overlooked.

The following guideline can be formulated: Take all the necessary actions to ensure that the details of operation specified at the calling context for a routine are preserved for as long as they will be needed. The necessary actions could include preserving certain state information that accumulates beyond the context of the original call, if it is essential to the proper completion of the original operation. In the example just given, the value of `>IN` (the position in the input stream for block 81 where interpre-

tation last stopped) is part of the critical information that must be preserved. So both the original LOAD parameter (81) and the value of >IN must be stack-buffered to guarantee their proper restoration later as part of the continuing task of interpreting block 81.

In the case of recursive or reentrant routines, the routine-to-routine interface involves several executing instances of the same routine. Newer LOAD calls do not walk all over older LOAD calls. Through the stack buffering it performs, LOAD synchronizes the current execution instance with any previous execution instances.

Too Many Calling Contexts with Different Requirements

The F83 vocabulary mechanism appears to be subject to losses of synchronization. I believe this occurs because the vocabulary mechanism is overworked. There are too many calling contexts with different requirements.

Sometimes the calling context for an F83 vocabulary change is an explicit vocabulary switch entered by the programmer (such as entering ONLY FORTH). At other times, the calling context is a word that is setting up special environmental modes, such as an editing mode or a CODE compilation mode.

(The system needs occasional tweaking, but only under unusual conditions. Still, the goal must be to build Forth systems with the fewest possible "gotchas.")

The code for F83 vocabularies is not poor code, but the design can be questioned. The design can be criticized because it depends on code in far-flung locations for its synchronization. For example, the colon and semicolon routines contain vocabulary operations. Such code seems out of place in those locations. Similarly, no location seems to be a good place to put any corrective ("fully synchronizing") code.

Choosing Between Interface Options

A loss of refined control can arise due to the use of too few discrete control elements compared to operations that are sought. Combining an automobile's control elements for braking and acceleration might at first seem to simplify a car's operation. However, certain operations may be become difficult, such as allowing the car to slow down by coasting. With separate pedals for acceleration and braking, letting the car coast is performed with very little effort.

Being able to select the best control elements is half of the battle of good interface design.

In any case, you don't want to end up with too few or too many control elements. To consider various options, I often use trial and error so I can get a feel for the different approaches.

For application domains that are new to us, we often have too little experience to render stable judgments. Today, one approach feels best; tomorrow, a different approach tempts us. (Nothing is more humbling than to be the primary user of your own software creations over a period of many years.)

(Continues on page 19.)

Product Watch

APRIL

Triangle Digital Services Ltd. announced a half-price reduction in the quantity price for its TDS9092, down to £50 each. The TDS9092 is an eight-bit control computer based on a surface-mount microprocessor with on-board Forth. A custom gate array provides a watchdog timer, character and graphics LCD interfaces, more parallel ports and spare address decoding. Features of the Forth board include two timers, two serial ports, and support for I^2C peripherals. Software support is included for 32-bit math, trigonometry. The development system Starter Pack is also halved in price, down to £150. It adds PC software, non-volatile RAM, and disk-based library routines. For £10, the 275-page manual from the Starter Pack can be purchased separately. It includes circuit diagrams for LCD, keypad, and stepper motor interfaces. (A 16-bit version, the TDS2020 remains available as well.)

APRIL

AM Research announced several new products. Leading the pack is the 80C537-based amr537LC. The processor is comparable to many 16-bit processors, but is code-compatible with the 8051. It uses low-power CMOS and runs at speeds up to 16 MHz. The new SBC includes 10-bit A/D with 12 inputs, timer-counters, two UARTs. The amr51LC is the most versatile new SBC in the lineup. Some configurations have an I^2C port or internal A/D, while others have large ROM spaces, extra timer-counters, EEPROM, or extra RAM. A new version of amr8051 Forth is also available, which previous customers can download from the AM Research bulletin board.

A new product line was also introduced around the MC68HC11L1. The amr8051 development system has been re-implemented for this new family of SBCs. One of the innovations in these new SBCs is a serial Boot Loader, which eliminates the need for EPROM or ROM during development, yet acts like a production, ROM-based system.

COMPANIES MENTIONED

Triangle Digital Services Ltd.
223 Lea Bridge Road
London E10 7NE England
Fax: 081-558 8110
Phone: 081-539 0285

AM Research
4600 Hidden Oaks Lane
Loomis, CA 95650
Fax: 916-652-6642
Phone: 916-652-7472

The European Forth Conference, EuroForth'94

Exploiting Forth: Professionally, Commercially, & Industrially

EuroForth, the annual European Forth Conference is celebrating its tenth anniversary this year in England. The Conference title, "Exploiting Forth", reflects the need in todays economic climate to make the best use of all the features that Forth provides. In particular this year's conference will show all the benefits and capabilities of merging Forth with modern programming environments.

The EuroForth conference provides a forum for the exchange of techniques, philosophies, and application notes, with papers presented by a range of speakers from industry, commerce, and academia. The conference covers both software and hardware, including stack-based processor architectures.

Delegates from all parts of Europe including Eastern Europe and the Former Soviet Union are expected. EuroForth is an international conference with delegates from other continents. EuroForth is a friendly conference at which time is made available for meeting people, for informal discussions, and for contacts.

Topics and Papers

The subjects proposed are shown below, together with subjects covered by papers already received.

Working with GUIs
Objects, Natural Languages, and Databases
Formal Methods
Programming Techniques
Forth Hardware
Compiler Construction
Commercial Topics and Packages
Industrial Applications
Networking and Communications

Conference delegates are welcome and encouraged to give papers on subjects related to the conference topics. These papers should be no more than 6 pages. Suggestions for any topic not listed will be gladly considered. Papers should take between 20 and 25 minutes to deliver including questions.

Abstracts should be submitted as soon as possible for acceptance by the committee. Refereed papers must be in by May 31st. Camera ready copy is required by 10 October 1994, so that delegates can receive the papers at the conference. Late papers may be accepted at the discretion of the committee.

Proceedings

These will be published by January 1995, and will be available from the Conference Organizer, and other sources of Forth literature. They will also be published by the Forth Interest Group.

Exhibitions and Demonstrations

The conference includes an exhibition and demonstration session at which systems discussed during the other sessions will be on show, alongside commercial products including Forth engines.

Location and accommodation

EuroForth'94 is being held in a pleasant hotel situated in the heart of Winchester, a lovely medieval city featuring the well known Winchester Cathedral. The city is only a 50 minute train journey from London to Winchester, with easy access from both Gatwick and Heathrow airports. Many activities can be found less than 25 miles away including:

Broadlands - The house of the late Lord Mountbatten, National Motor Museum, Jane Austin's House, Stonehenge, Salisbury Cathedral, Mary Rose Ship Hall & Exhibition, HMS Victory - HM Naval Base Portsmouth, The New Forest

Saturday night of this conference is Guy Fawkes night in England. This is celebrated in the town of Winchester with a candle lit procession and fireworks - it will be a night to remember.

Sunday night there will be an informal survivors party at MPE in Southampton for those staying until Monday.

Fees

Resident delegate	£290.00 + VAT.	(including conference fee, hotel accommodation, and all meals)
Nonresident delegate	£215.00 + VAT.	(includes conference fee, lunch on Friday, Saturday, and Sunday
Resident visitor	£140.00 + VAT.	(including shared accommodation and all meals.)
Student Rate	£192.00 + VAT.	(To obtain this rate you must have a National Union of Students card. Also accommodation will be shared.)

VAT - Please note that VAT (sales tax) is charged at 17.5% in addition to the above prices.

Registration: For further information please contact:

The Conference Organizer, EuroForth'94
c\o Microprocessor Engineering Limited
133 Hill Lane, Southampton SO1 5AF, England
Tel: +44 703 631441, Fax: +44 703 339691
net: mpe@cix.compulink.co.uk